2007-12

A model for minimizing numeric function generator complexity and delay

Knudstrup, Timothy A.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A MODEL FOR MINIMIZING NUMERIC FUNCTION GENERATOR COMPLEXITY AND DELAY

by

Timothy A. Knudstrup

December 2007

Thesis Advisor:	Jon T. Butler
Co-Advisor:	Chris L. Frenzen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A Model for Minimizing Numeric Function Generator Complexity and Delay		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Timothy A. Knudstrup			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency Fort Meade, MD		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE** A	

13. ABSTRACT (maximum 200 words)

Numeric Function Generators (NFGs) allow computation of difficult mathematical functions in less time and with less hardware than commonly employed methods. They compute piecewise linear (or quadratic) approximations that represent the value of the specified function for a given input value. The domain of the function is divided into enough segments so that the approximation is within the required error to the exact value.

NFG hardware complexity and delay depend on the number of segments required, the arithmetic devices used to approximate the function, and the number of bits used to represent the numbers being calculated. This thesis develops an accurate method to quantify hardware complexity and delay for various NFG configurations implemented on a Field-Programmable Gate Arrays (FPGAs). The algorithms and estimation techniques apply to various NFG architectures and mathematical functions.

This thesis compares hardware utilization and propagation delay for various NFG architectures, mathematical functions, word widths, and segmentation methods. It shows that quadratic NFGs perform better than linear NFGs when the precision is above a threshold. It also shows that the majority of the functions in our benchmark have lower complexity when uniform segmentation is implemented. A criterion for choosing a segmentation method is shown for specific cases.

14. SUBJECT TERMS Numeric Function Generators, Hardware Complexity, Field-Programmable Gate Arrays (FPGAs), Computer Arithmetic, Linear Approximations, Quadratic Approximations, Segment Index Encoders, Segmentation Algorithms			**15. NUMBER OF PAGES** 236
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A MODEL FOR MINIMIZING NUMERIC FUNCTION GENERATOR COMPLEXITY AND DELAY

Timothy A. Knudstrup
Lieutenant, United States Navy
BSEE, University of Texas, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2007

Author: Timothy A. Knudstrup

Approved by: Jon T. Butler
Thesis Advisor

Chris L. Frenzen
Co-Advisor

Professor Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Numeric Function Generators (NFGs) have allowed computation of difficult mathematical functions in less time and with less hardware than commonly employed methods. They compute piecewise linear (or quadratic) approximations that represent the value of the original function for a given input value. The domain of the NFG is divided into enough segments such that the approximation is within the required error to the actual value of the function. The linear (or quadratic) approximation varies for each segment. The overall hardware complexity and propagation delay depend on the number of segments required, the arithmetic devices used to approximate the function, and the number of bits used to represent the numbers being calculated.

This thesis develops an accurate method to quantify hardware utilization and propagation delay for various NFG configurations implemented on Field-Programmable Gate Arrays (FPGAs). The algorithms and estimation techniques apply to different NFG architectures and to different mathematical functions. This thesis compares hardware utilization and propagation delay for various NFG architectures, mathematical functions, word widths, and segmentation methods. It shows when a quadratic NFG requires less hardware and when it has a longer delay than its linear NFG counterpart for various functions. It also establishes a criterion for when non-uniform segmentation is beneficial for any function, based on the size of the NFG. The findings in this thesis show that NFGs with non-uniform segmentation generally require more hardware and almost always have longer delays than NFGs with uniform segmentation. They also show that quadratic NFGs required less hardware and have shorter delays as the size of the NFG gets larger.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BKM	Bajard, Kla, Muller algorithm
BRAM	Block Random Access Memory
BUP	BRAM Utilization Percentage
CLAH	Carry Look Ahead Adder
CORDIC	Coordinate Rotation Digital Computer
DCM	Digital Clock Manager
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HUP	Hardware Utilization Percentage
I/O	Input / Output
IOB	I/O Block
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Software Environment
LSB	Least Significant Bit
LN	Linear NFG with Non-uniform Segmentation
LNB	LN with a Basic Architecture
LNC	LN with a Compact Architecture
LU	Linear NFG with Uniform Segmentation
LUB	LU with a Basic Architecture
LUC	LU with a Compact Architecture
LUT	Look-Up Table
MSB	Most Significant Bit
MULT18x18	18 by 18 bit Signed Multiplier on Xilinx Virtex-II XC2V6000 FPGA
MUP	MULT18x18 Utilization Percentage
MUX	Multiplexer
NFG	Numeric Function Generator
NPS	Naval Postgraduate School
OBM	On-Board Memory
PP	Partial Product

PPG	PP Generator
QN	Quadratic NFG with Non-uniform Segmentation
QNB	QN with a Basic Architecture
QNC	QN with a Compact Architecture
QU	Quadratic NFG with Uniform Segmentation
QUB	QU with a Basic Architecture
QUC	QU with a Compact Architecture
RAM	Random Access Memory
ROM	Read Only Memory
SIE	Segment Index Encoder
SRC	Seymour R Cray
SUP	Slice Utilization Percentage
USN	United States Navy
Verilog	A C-Based HDL
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

EXECUTIVE SUMMARY

This thesis describes a complexity/delay analysis of numeric function generators (NFGs) used in high-speed circuits for realizing arithmetic functions like $f(x) = \sin(x)$, $f(x) = \ln x$, $f(x) = \sqrt{-\ln x}$, etc. Specifically, it shows how complexities and delays for NFGs can be estimated without having to build the circuit. It begins by constructing basic arithmetic components that are often used in NFGs. Each component is analyzed in depth to estimate its complexity and delay based on the number of input bits, *n*. Models of common NFGs are built realizing an approximation equation, $y(x)$. The models are used to compare various NFG architectures for particular functions. NFGs with linear approximation equations are compared to NFGs with quadratic approximations.

Uniform and non-uniform segmentation methods are also compared in this thesis because the complexity and delay of an NFG greatly depends on the complexity and delay of its coefficients table and associated segment index encoder (SIE). Uniform segmentation divides the function interval into $s_{\min}^{unif}$ segments of even width, while non-uniform segmentation divides the interval into $s_{\min}^{non-unif}$ segments of varying widths. The maximum segment width is determined by a maximum allowable error ε, where $\varepsilon = |f(x) - y(x)|$. Non-uniform NFGs always require fewer segments than uniform NFGs, but they also require an SIE in order to determine within which segment ***x*** lies.

For 13 of the 15 functions analyzed in this thesis, non-uniform segmentation offers no benefits. However, when non-uniform segmentation drastically reduces the number of segments in an NFG, it can reduce the overall hardware complexity. This occurs in the remaining 2 functions. The amount of reduction from uniform to non-uniform segmentation can be expressed as a ratio, namely the *segment reduction ratio* (*SRR*). The minimum *SRR* required in order for non-uniform segmentation to be beneficial is SRR_{crit}. SRR_{crit} depends on the number of segments, *s*, which depends on ε and the properties and domain of the function being realized. This thesis also shows that the *SRR* of a given function depends only on the properties of that function and its

domain. Thus, for a given function $f(x)$, when $SRR_{f(x)} < SRR_{crit}(n,s)$, then an NFG with non-uniform segmentation requires less hardware than the same NFG with uniform segmentation. When the number of segments (corresponding to the number of memory locations) is restricted to a power of two, the number of segments for non-uniform segmentation is $s^{non-unif} = 2^{\lceil \log_2 s_{\min}^{non-unif} \rceil}$ and number of segments for uniform segmentation is $s^{unif} = 2^{\lceil \log_2 s_{\min}^{unif} \rceil}$ and $SRR_{crit,\min}$ becomes a function only of *n*.

Therefore, for a basic linear NFG, if $\frac{\int_a^b \sqrt{\left| f^{(2)}(x^*) \right|} dx}{(b-a)\sqrt{\left| f^{(2)}(x^*) \right|}} < \frac{4}{n+4}$ (or $SRR^{\text{Basic Linear}} < SSR_{crit,\min}^{\text{Basic Linear}}$), then non-uniform segmentation yields a smaller amount of hardware. This is true for basic quadratic NFGs when $\frac{\int_a^b \sqrt{\left| f^{(3)}(x^*) \right|} dx}{(b-a)\sqrt{\left| f^{(3)}(x^*) \right|}} < \frac{6}{n+6}$. From these equations, a critical value of *n* can be determined, n_{crit}, below which it is always more hardware efficient to use non-uniform segmentation. The derivations of these equations assume that LUT cascades are used in the SIE and Chebyshev polynomials are used to determine the coefficients for the approximation equations. They also assume that basic NFG architectures are used. The term "basic" refers to an architecture that does not truncate bits during its arithmetic operations.

This thesis shows that non-uniform segmentation always has a longer delay than uniform segmentation, except in rare trivial NFGs (where $n \le 8$). In fact, when NFG architectures for 15 functions were compared in terms of delay, non-uniform NFGs proved the best only in a few cases when $n \le 2$. If $n \le 2$, then an NFG is not required since two LUTs can be used instead. Appendices D.2.2 and D.3.2 show the best architectures based on delay for 15 functions.

Linear and quadratic NFGs are also compared in this thesis. Estimation results show that linear NFGs consume less hardware than quadratic NFGs for *n* less than ≈ 25 to 29 bits (for the 15 functions compared). They also have smaller delays than quadratic

NFGs for $n \approx 37$ to 39 bits. This thesis shows which of the four basic architectures (linear uniform (LUB), linear non-uniform (LNB), quadratic uniform (QUB), quadratic non-uniform(QNB)) is best in terms of hardware utilization and delay for all 15 functions analyzed. It also shows the best of four compact NFG architectures (LUC, LNC, QUC, and QNC). The compact architectures are similar to the basic architectures except they require smaller arithmetic units.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM DEFINITION

Computer calculations of numerical functions are required in many applications ranging from computer graphics to robotics to radar return processing [11]. Trigonometric, logarithmic, exponential, and power functions are all widely used, as well as combinations of them. Well designed application specific integrated circuits (ASICs) generally offer the fastest computation time for a specific function because they are designed with that function in mind. Therefore, they are usually expensive because they are not in high demand. However, they typically serve only one purpose. Reconfigurable computers are an important developing technology that can be used to perform specific computations. They provide a universal platform for a wide variety of tasks and allow the task to be changed. Reconfigurable computers often use Field-Programmable Gate Arrays (FPGAs) to implement the desired logic designs. The benefit of using FPGAs for complex computations is that the FPGA can perform the computations while the processor performs other system-related tasks. Having the FPGA compute the desired function is generally faster than having the main microprocessor do the same computation. The main processor can also perform other systems tasks instead, therefore making the entire computer system faster.

This thesis analyzes methods for approximating numerical functions. It also discusses the implementations on FPGAs, so problem solutions must be able to fit on a particular FPGA while still meeting the speed and precision requirements of the application requiring the function computation. This section discusses some of the hardware configurations that are currently employed in performing these calculations, including using numeric function generators (NFGs).

1. Methods for Numeric Function Computation

There are several methods for computing real functions with electronic hardware. The following methods are commonly employed.

a. Lookup Table

A simple method for computing a numerical function is by using a lookup-table (LUT). LUTs use input variable ***x*** as the address to a memory block. The data word stored at that address is the function's value $f(x)$. This method requires an enormous amount of memory for any relatively large computing system. Consider a simple architecture where ***x*** has 16 bits and the result has 16-bits. The LUT requires $2^{16} \times 16 = 2^{20} = 1,048,576$ memory bits, or 131,072 bytes. This is relatively large amount for such a small number system, making it very difficult to implement on FPGAs. Modern computer systems require *n* to be much larger, generally 32 or 64-bits. A 32-bit LUT requires over 17 Gbytes, and a 64-bit LUT requires 1.5×10^{20} bytes. Because of the size requirements, LUTs are generally not the best solution for reconfigurable computers because they do not fit on commonly used FPGAs.

b. CORDIC

COordinate Rotational DIgital Computer (CORDIC) algorithms are often used because they require a small amount of hardware [1] [11]. They are used in many pocket calculators and floating-point coprocessors [6].

CORDIC devices perform successive arithmetic operations iteratively. Each of the iterations increases the precision of the result. Modern technology requires a high accuracy in very little time. Since the precision of CORDIC algorithms are proportional to the computation time, they are becoming less acceptable [16] for high-speed applications. In addition, CORDIC algorithms have been developed only for a limited set of functions.

c. Power Series

Some numerical functions can be decomposed into an infinite series known as a power series. The power series is an infinite sum of powers of an input variable ***x***, or $f(x) = \sum_{i=0}^{\infty} a_i (x-c)^i = a_0 + a_1(x-c) + a_2(x-c)^2 + \ldots$. When *c*=0, this

architecture can be implemented compactly in an iterative form, requiring a multiplier, an adder, a register, and memory storage for the coefficients a_i. Like the CORDIC algorithm, the accuracy of the result depends on the number of iterations of the algorithm and it can be applied only to a limited number of functions. For example, $f(x) = e^x$ can be calculated by represented by the power series $e^x = \sum_{i=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$ but more complex functions might not be able to be computed.

d. Shift and Add Algorithms

Shift and add algorithms, such as the BKM algorithm [6] (named for its developers J.C. Bajard, S. Kla, and J.M. Muller), have been developed to compute functions without using multipliers. They simply iterate shifts and add, thus reducing the hardware significantly. BKM algorithms compute a limited number of functions, including: 2-D vector rotations, logarithmic functions, exponential functions, *sine* and *cosine* functions and *arctan* functions [6]. However, their precision still depends on the number of iterations in the computation; therefore they often do not meet the requirements of high-speed applications.

e. NFGs

NFGs return a function value by using piece-wise approximations. NFGs require a few basic arithmetic devices and a coefficient memory or LUT. The memory size generally depends on the function being implemented and the precision of the system, but it is always smaller than that of using a LUT alone. NFGs perform the same numerical calculations for every function (for example, $f(x) = c_1 x + c_0$ for linear approximation), but just use different coefficients. NFGs can be considered a combination of the methods described above. They use less memory than a LUT alone and they often employ arithmetic devices (multipliers and adders) similar to power series architectures. However, the computation by an NFG is not iterative. Thus, NFGs can compute any function with a small amount of hardware and a small computation time.

2. Goal of This Thesis

This thesis analyzes NFG architectures in depth to make accurate estimations of complexity and delay. In this way, we can understand easily, for example, how tradeoffs can be made between complexity, delay and accuracy. The only other way is to build actual designs, which is computationally intensive. It analyzes and compares arithmetic component complexity and delay as well as NFG architectures that are composed of those components. It develops models of common architectures, and provides a framework with which any architecture can be built. Models for simple NFG architectures are compared to determine which are the most efficient with respect to hardware utilization and delay. Comparisons include hardware utilization and delay for linear versus quadratic NFGs, as well for NFGs with uniform versus non-uniform segmentation.

B. THESIS ORGANIZATION

Chapter I introduces the problem being discussed in this thesis, including some of the current methods to solve the problem. It also discusses why this thesis focuses on NFGs instead of analyzing the other methods. Chapter II focuses on the basic understanding of how linear and quadratic NFGs work, including their basic architectures. Chapter III develops accurate tools to measure hardware utilization and propagation delay for the basic arithmetic components commonly used in NFGs. It explains how simulation data was obtained and used to estimate various NFG configurations. Chapter IV builds models for NFG architectures commonly used in recent resources. Each model can realize any function. Chapter IV also establishes a framework by which any particular NFG architecture can be built. Chapter V compares the models in Chapter IV for example functions. It shows when it is better to use quadratic versus linear NFGs for several functions based on hardware utilization and delay. It also develops a criterion for determining whether or not it is better to use non-uniform segmentation. Chapter VI summarizes the findings of Chapter V and discusses future applications of the modeling methods in this thesis.

II. BACKGROUND ON NFGS

This chapter discusses how linear and quadratic NFGs operate. It is mostly concerned with NFGs implemented on reconfigurable computers and FPGAs. The architecture of an NFG is somewhat independent of the function being realized. Thus, a generic NFG can be used to realize a wide range of functions without having to redesign logic circuits. Also, NFGs on FPGAs are reconfigurable, so it is easy to reprogram it to compute a different function.

A. GENERAL NFG OPERATION

An NFG is an arithmetic logic device that estimates the value of a real function $f(x)$ for a given input $\boldsymbol{x}$ using a piecewise approximation $y(x)$. The domain of the NFG $[a,b]$ is divided into s segments each with domain $[x_{\min,i}, x_{\max,i})$, where i is the segment index number. Thus, $y(x) = y_i(x)$ iff $x_{\min,i} \le x < x_{\max,i}$. Each approximation function $y_i(x)$ may be a linear, quadratic or some other simple function of $\boldsymbol{x}$. For all inputs $\boldsymbol{x}$, the NFG must determine what segment it is in in order to determine the approximation function.

B. LINEAR NFGS

Simple linear NFGs use the approximation function $y_i(x) = c_{1i}x + c_{0i}$ for each segment, where $i \in$ and $1 \le i \le s$. The values for c_{1i} and c_{0i} are stored in a coefficients table and recalled once the segment number i is known for a particular $\boldsymbol{x}$. Figure 1 shows an example of how linear approximation functions are used for each segment. In the example, $f(x) = 2^x$ with a domain [0,5] and s=5, and the particular segment index i=4.

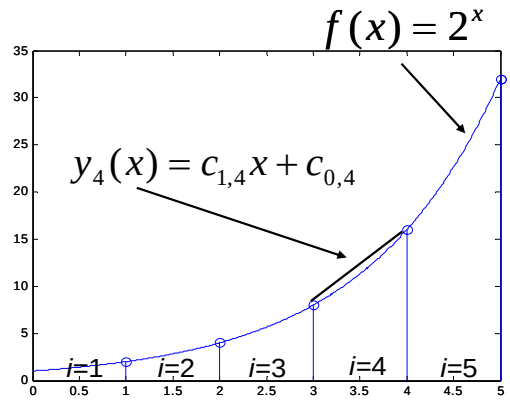
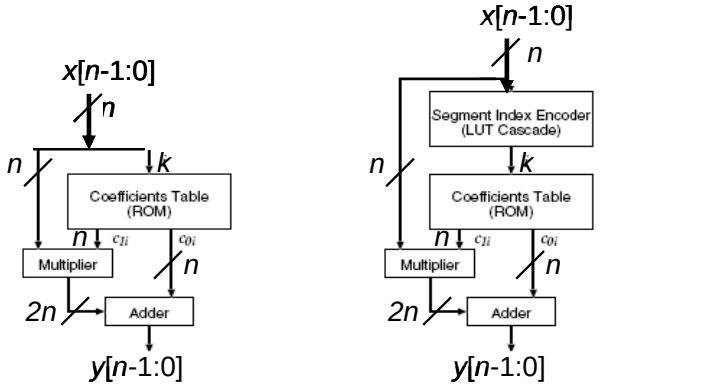


Figure 1 Linear Approximation for a Single Segment for $f(x) = 2^x$.

1. Basic Linear NFG Architecture

The architecture of a basic linear NFG is shown in Figure 2. It consists of arithmetic components (multiplier and adder), a memory to store coefficients, and logic circuit to determine the segment index (if necessary).



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 2 Basic Linear NFG Architecture. (After [12])

2. Approximation Techniques

The linear equations $y_i(x)$ are computed prior to constructing the NFG for each segment. They are stored in the coefficients table. The coefficients can be determined by several methods, a few of which are described below.

a. Secant Line Approximation (SLA)

For a given segment *i*, the endpoints of the segment ($x_{\min,i}$ and $x_{\max,i}$) are used to determine the slope and intercept values (c_{1i} and c_{0i} , respectively). The slope is $c_{1i} = \frac{f(x_{\max,i}) - f(x_{\min,i})}{x_{\max,i} - x_{\min,i}}$ and the intercept value is $c_{0i} = f(x_{\min,i}) - c_{1i} x_{\min,i}$. The error of this approximation is $\varepsilon_{SLA} = \left| f(x) - y_i(x) \right|_{\max}$.

b. Modified Secant Line Approximation (MSLA)

The SLA method is a quick method to estimate a function over a given segment, but it is obviously not the most accurate. The maximum error in a particular segment can be reduced by adjusting c_{0i} by a value less than ε_{SLA} . Consider a function $f(x)$ that is monotone increasing or decreasing over $[x_{\min,i}, x_{\max,i}]$. The linear approximation $y_i(x) = c_{1i}x + c_{0i} \neq f(x)$ on $\left(x_{\min,i}, x_{\max,i}\right)$. Therefore, $y_i(x)$ is always greater than or less than $f(x)$ on $\left(x_{\min,i}, x_{\max,i}\right)$. If $y_i(x) > f(x)$ on $\left(x_{\min,i}, x_{\max,i}\right)$, then subtracting $\varepsilon_{sla}/2$ from c_{0i} (from the SLA), yields a maximum error of $\varepsilon_{MSLA} = \varepsilon_{SLA}/2$ for the segment. Figure 3 shows the difference between the linear approximation equations using SLA and MSLA.

c. Least Squares Approximations

MATLAB uses a function called polyfit to calculate coefficients for linear, quadratic and higher order approximation functions based on the least-squares error. The least squares method is commonly used to minimize the sum of the differences between two given functions. This particular method is not desired for applications with NFGs. NFGs are concerned with being able to compute a value of a function and yield an answer that is correct to the limits of the number system on which it is implemented.

NFGs are designed to produce a result with an error that is less than a maximum specified error, and not to minimize the sum or average errors. The example in Figure 3 shows that the polyfit function (using a linear fit) produces a larger maximum error than the MSLA.

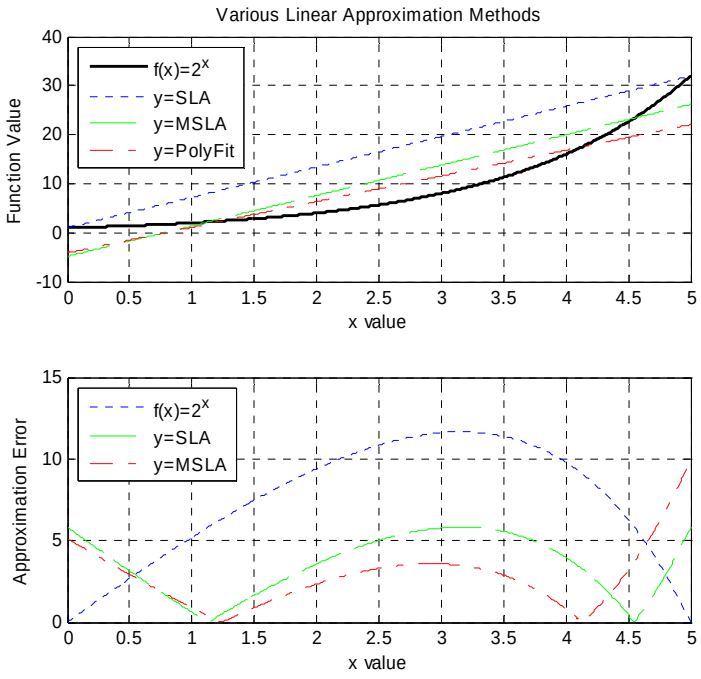


Figure 3 Linear Approximations of $f(x) = 2^x$.

C. QUADRATIC NFGS

Quadratic NFGs use the approximation function $y_i(x) = c_{2i}x^2 + c_{1i}x + c_{0i}$ for each segment, where $i \in$ □ and $1 \le i \le s$. The values for c_{2i}, c_{1i} and c_{0i} are stored in a coefficients table and recalled once the segment number is known for a particular ***x***. Figure 4 shows an example of how quadratic approximation functions are used for each segment. The example is the same as discussed for a linear approximation above.

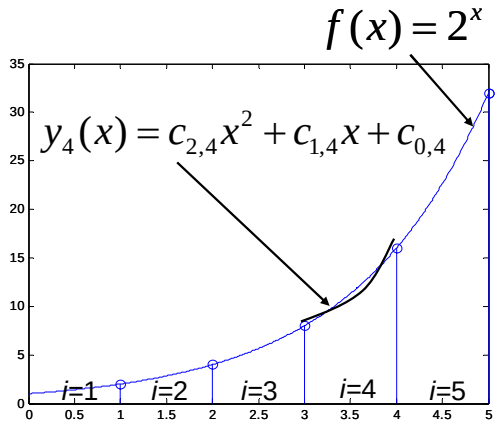
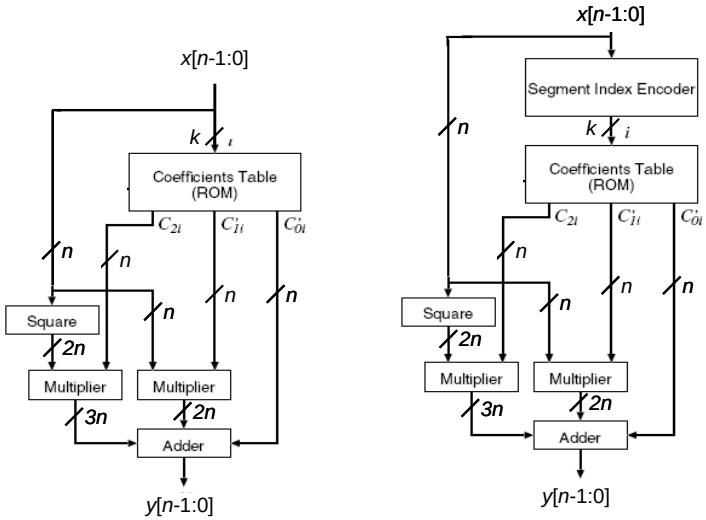


Figure 4 Quadratic Approximation for a Single Segment for $f(x)=2^x$.

1. Basic Quadratic NFG Architecture

The architecture of a basic quadratic NFG is shown in Figure 5. Like the linear architecture, it also consists of arithmetic components (multipliers and adders), a memory to store coefficients, and logic circuit to determine the segment index. However, quadratic NFGs require three multipliers and a 3-input adder. Although quadratic NFGs require more arithmetic devices than linear NFGs, they require fewer segments, and thus smaller memory sizes.



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 5 Basic Quadratic NFG Architectures. (After [8]).

2. Approximation Techniques

Determining the best coefficients for quadratic approximations is quite difficult and cannot be generalized for all functions. However, some methods have been considered sufficient to find coefficients that can accurately approximate given functions. Several approximation techniques are outlined in [6], but the ones of concern are those that minimize the maximum error in each segment. These are known as the least maximum polynomial approximations [6].

a. 2nd Order Chebyshev Polynomial Approximation

Chebyshev polynomials provide a straightforward method for determining the coefficients required to approximate a function with any order polynomial. "Chebyshev polynomials play a central role in approximation theory [6]." They have been studied in depth and have many properties that allow simple error calculations. Their properties are used to prove asymptotic relations for finding the widest segment required and for finding the minimum number of segments required.

b. Minimax Approximation

Second order minimax approximations use the fact that there are at least four values of x where the maximum approximation error is reached with alternating signs, namely x_0, x_1, x_2, and x_3 [6]. The minimax approximation solves the following set of equations to determine the coefficients of the polynomial approximation $y_i(x) = c_{2i}x^2 + c_{1i}x + c_{0i}$.

$$
\begin{aligned}
y(x_0) - f(x_0) &= \varepsilon \\
y(x_1) - f(x_1) &= -\varepsilon \\
y(x_2) - f(x_2) &= \varepsilon \\
y(x_3) - f(x_3) &= -\varepsilon \\
\frac{dy(x_1)}{dx} - \frac{df(x_1)}{dx} &= 0 \\
\frac{dy(x_2)}{dx} - \frac{df(x_2)}{dx} &= 0
\end{aligned}
$$

c. Remez Algorithm

The Remez algorithm for finding polynomial coefficients is an iterative method that starts with coefficient value estimates from typically either Chebyshev or minimax approximations. The points where the error is maximum are found and then used to calculate new coefficients, reducing the new error. Since Chebyshev polynomials have approximations that are very close to optimum, the Remez algorithm quickly converges. This method often provides coefficients that more accurately compute the NFG approximations. This results in larger segment sizes. Therefore, it also results in fewer required segments.

D. FACTORS CONTRIBUTING TO COMPLEXITY AND DELAY

The complexity and delay of an NFG depends on the complexity and delay of its arithmetic components, as well as the size of the coefficient table required.

1. Factors Affecting Arithmetic Component Complexity and Delay

a. The Size of the NFG

The size of the NFG *n*, refers to the number of bits input into the NFG. The examples analyzed in this thesis also assume that the NFG produces the same number of bits for its result. As *n* grows, the complexity and delay grow because more logic gates are required for each of the components in the NFG. For example, a 32-bit adder requires more logic gates and has a longer delay than a 16-bit adder.

b. NFG Architecture

NFGs can be configured in several ways. The architecture determines what components and how many components are needed to realize a function $f(x)$. For example, a basic linear NFG with uniform segmentation requires a multiplier, adder, and coefficients table. An equivalent basic quadratic NFG with uniform segmentation requires three multipliers and two adders. Other configurations can require other arrangements and numbers of component which all contribute to the total complexity and

delay. Some NFGs can be arranged to compute several operations in parallel to minimize overall delay. Thus, the architecture plays a large role in the complexity and delay of the NFG components.

2. Factors Affecting the Number of Segments

The number of segments depends on the size of the NFG, *n*, $f(x)$ and its domain [*a*,*b*], and the segmentation method. The number of segments determines how much memory is required to store the coefficients for the estimation equation $y(x)$. They are analyzed further in later chapters.

a. Function and NFG Domain

Asymptotic equations in [5] show that the minimum segment width required is a function of the 2nd or 3rd derivative of $f(x)$ for linear and quadratic NFGs respectively. Thus, for a given NFG domain, the number of segments required also depends on the particular function $f(x)$ realized by the NFG. As the domain of the NFG gets larger, more segments are required for the same allowable error ε.

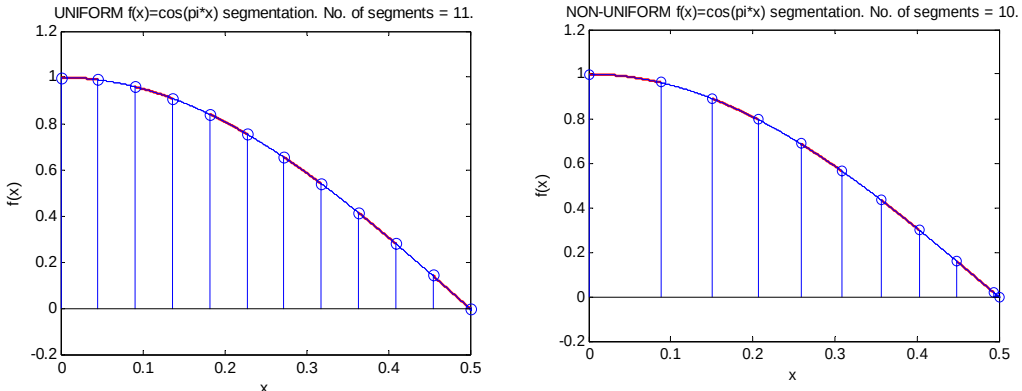
b. The Size of the NFG

The number system, or the number of bits in the input and output of an NFG, plays a role in determining the maximum allowable error. The goal of an NFG is to compute an approximation with an error that won't be noticed by the system that is using the NFG. As *n* grows the allowable error ε gets smaller, requiring more segments. Also, the size of the NFG generally affects the required precision for the NFG, which affects the number of required segments. Therefore, the size of the coefficient table also depends on *n.*

c. Segmentation Method

Choosing between uniform and non-uniform segmentation can drastically affect the overall number of segments required. Methods in [5] derive a minimum

segment width $\sigma_{\min}$, for a given function on a given interval [*a*,*b*]. Dividing the interval into uniform-width segments, each $\sigma_i = \sigma_{\min}$ for all *i*, where $1 \le i \le s_{\min}$. Here $s_{\min}$ is the minimum number of segments required and $s_{\min} = \frac{b-a}{\sigma_{\min}}$. Non-uniform segmentation over the same interval first finds $\sigma_{\min}$ and uses it for a particular segment, σ_i. For optimum segmentation, a new $\sigma_{\min}$ is found for the remaining portion of the interval (excluding segment *i*). This occurs repeatedly until the segments include the entire domain of the NFG. Non-uniform segmentation always produces fewer segments. Figure 6 shows an example to compare the number of segments required for uniform and non-uniform segmentation of $f(x) = \cos \pi x$ on [0,0.5] for an $\varepsilon = 2^{-9}$. Uniform segmentation requires 11 segments, and non-uniform segmentation requires 10.



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 6 Uniform vs. Non-Uniform Segmentation. (From [20])

E. CHAPTER SUMMARY

This chapter shows how NFGs approximate real functions, including several methods for computing the coefficients of the approximation equations. It also shows factors that affect the complexity and delay of NFGs and the components required to construct four basic NFG architectures. The next chapter shows how each of these components (and others) can be built on the Xilinx Virtex-II. It estimates the complexity and delay based on the size of each component using simulation data and approximated data.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ANALYZING HARDWARE COMPLEXITIES AND PROPAGATION DELAYS

This chapter proposes a method to estimate circuit complexity and speed for common NFG components. This will allow us to compare the hardware complexity and speed of various NFG configurations. A standard method for measuring these quantities is proposed. The proposed method is applicable to a wide range of configurations, providing meaningful comparisons among various NFG configurations.

The supporting data was observed using particular hardware (Xilinx Virtex-II) and software (Xilinx ISE Project Navigator), but the methods can be applied universally to other FPGAs with minor alterations. Since the method of measuring is standardized, it provides a meaningful approach in understanding the relative complexity of realizing different arithmetic functions.

When actually designing an arithmetic logic device, pipelining can dramatically reduce propagation delays for the circuit. In best case scenarios, pipelining can cause the circuit to output an answer every clock period. A disadvantage of pipelining comes from an initial delay due to the pipeline depth. Large circuits tend to have a large pipeline depth, which means there is a long delay from the time data is input into the circuit, until the result comes out. Because pipelining can be implemented at a various points in a logic circuit, it is difficult to reach a standard way to measure time delay. For this reason, this thesis implements combinational logic circuits instead of pipelined circuits. In general, a combinational logic circuit that has a longer propagation delay will tend to have a longer pipeline depth as well. Thus, it is a relevant method of delay measurement.

A. HARDWARE RESOURCES

NFG component circuit designs are simulated and synthesized for the Xilinx Virtex-II XC2V6000 FPGA with a speed grade of -4. This is the FPGA that is presently available on the SRC-6, a reconfigurable computer at NPS. This section explains the general architecture of the Xilinx Virtex-II FPGA, including the available logic

resources. The Virtex-II includes Combinational Logic Blocks (CLBs), 18-by-18-bit signed multipliers (MULT18x18s), and Block Select RAM (BRAM). Figure 7 shows how these resources are arranged on the Virtex-II FPGA.

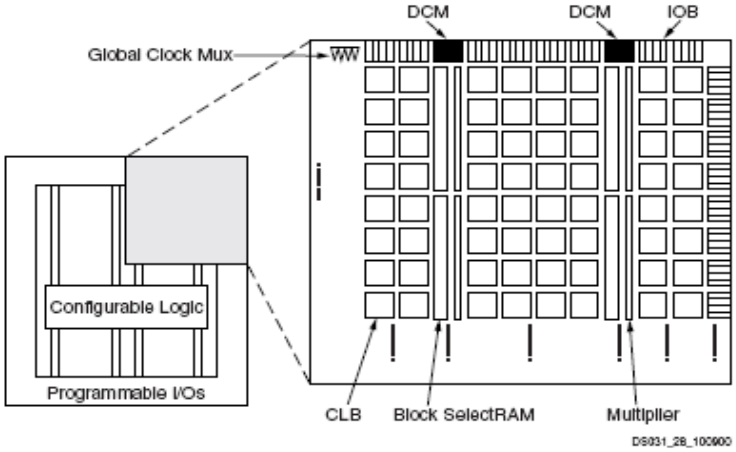


Figure 1: **Virtex-II Architecture Overview**

Figure 7 General Placement of Resources on Xilinx Virtex-II FPGA. (From [18])

Also shown are the Digital Clock Manager (DCM) units and Input/Output Blocks (IOBs), which are not used in the complexity measure. DCMs can be used to de-skew clock signals, manage multiple clock phases, create multiple frequency clock signals, and more [19]. The analyses in this thesis consider combinational logic delays and do not take into account complicated clocking schemes. Therefore, DCM usage is not considered in this thesis. IOBs route signals from the input pins to the logic circuitry in the FPGA and route signals from the logic circuitry to the output pins. The NFGs considered in this thesis are built from the available logic within the Virtex-II XC2V6000. Thus, for a given NFG size n, the number of IOBs consumed is $2n$. In this thesis, all of the available logic resources are always consumed before the IOBs. Therefore, the number of IOBs consumed is not relevant when comparing NFGs of the same size.

Each CLB on the Virtex-II FPGA is subdivided into four slices. Each slice is identical, except for its position in the CLB. Thus, the number of available slices is also a good measure of logic resources. Table 1 shows the five resources and the quantity

available on the Xilinx Virtex-II XC2V6000 FPGA. The amount of resources available, timing information for specific logic devices, and other specifications are included in the author's MATLAB file LoadISEDeviceData. It also imports some data from simulations. NFGs implemented on other FPGAs can be analyzed by altering LoadISEDeviceData to contain specifications for that particular FPGA.

Resource	**Quantity**
Slices	33792
MULT18x18	144
BRAM	144
IOB	1104
DCM	12

Table 1 Xilinx Virtex-II XC2V6000 Resources. (From [18])

1. CLBs

The most basic element of the CLB is the function generator. The function generator can be configured to realize a 4-input 1-output logic function or LUT, a ROM/RAM with 16 1-bit-words (16x1), or a 16 bit shift register. Even though 16x1 RAM units are realizable with a LUT, the circuits analyzed in this thesis do not require RAM, therefore there will be no further discussion of components that are related to RAM. For the purpose of this thesis, the function generator can be considered a look up table independent of what purpose it serves. For example, a 16x1 ROM is a 4 input to single output function. Xilinx has configured quick paths for linking these devices to larger configurations based on what purpose they serve. These timing characteristics are taken into account when building and analyzing the specific components on the FPGA. When considering how much hardware is used, the specific function of the function generator is irrelevant. The circuit designs in this thesis most often use the function

generator as a LUT. Therefore, in order to simplify terminology, each function generator is referred to as a LUT. Figure 8 illustrates a portion of the basic slice of a Virtex-II FPGA, highlighting some of the logic devices that are used in this thesis.

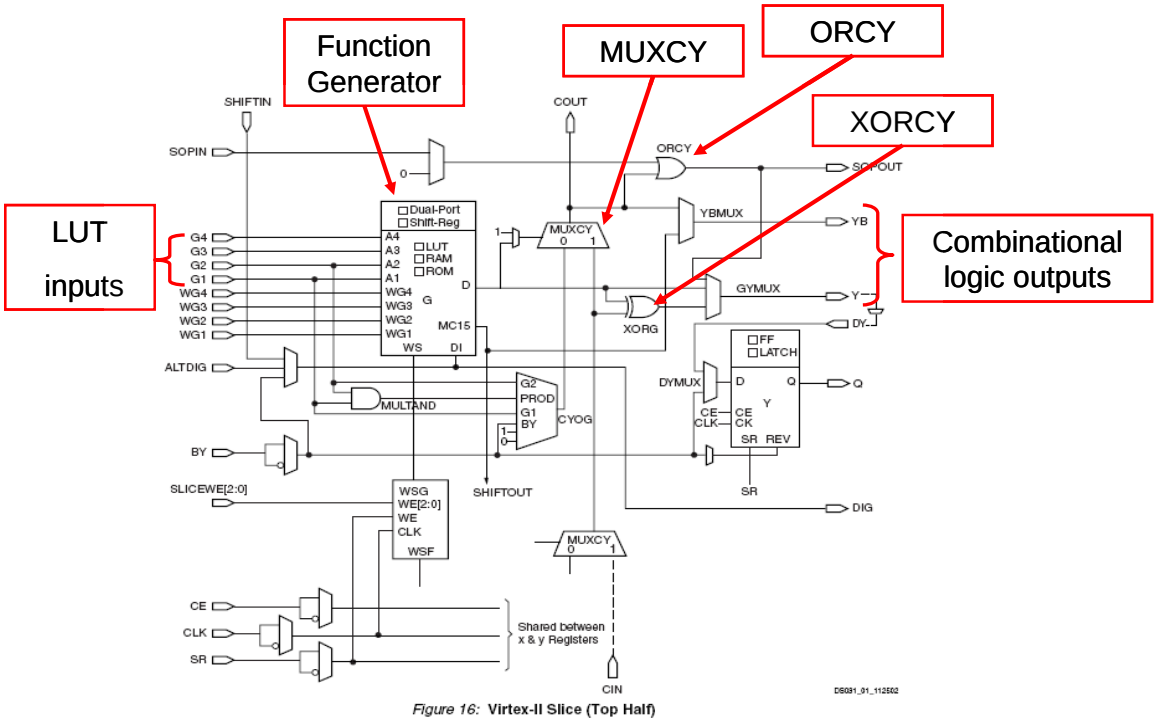


Figure 8 One-Half of a Xilinx Virtex-II Slice. (After [18])

A slice combines two LUTs with additional hardware including several MUXs, two clocked registers, and additional gates that are commonly used in arithmetic operations (XORCY, ORCY, etc.). Thus, Xilinx has made the basic slice extremely versatile and efficient for common operations. There are four slices per CLB. These are connected together efficiently with minimal signal propagation delay. Four slices comprise a CLB. See Figure 9 for an illustration of the CLB layout.

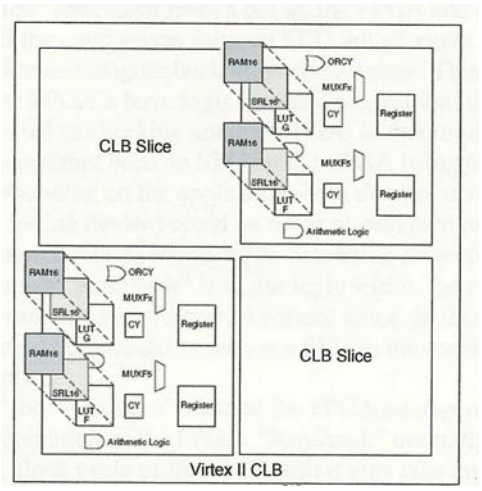


Figure 9 Xilinx XCV6000 CLB Layout. (From[18])

2. MULT18x18s

The MULT18x18 is a signed two's complement multiplier. Thus, it can multiply two 17-bit magnitude numbers, and return a 35-bit magnitude result along with an extra bit for the sign. The MULT18x18s are arranged in columns as shown in Figure 7. This reduces the propagation delay between the MULT18x18 and its surrounding components, allowing for fast connections between MULT18x18 to BRAMs, CLBs or IOBs. The MULT18x18s cannot be configured to perform other functions, but they may be used as multipliers with less than 18-bit multiplicands. There are a few benefits for using it for smaller multipliers. First, the circuit designer does not need to design a multiplier from CLBs (which would be slow). Second, because the multiplier does not consume CLBs, the CLBs can be used for other functions. This consumes all of the resources more evenly. Finally, when considering circuit performance, using multiplicands with fewer bits results in fewer bits in the product. This results in a smaller propagation delay through the MULT18x18. Xilinx has designed the Virtex-II such that the delay from the input to the output is linear with respect to the output pin. For example, if the MSB of the product comes off of pin α and it takes t_α to propagate through the MULT18x18, then a multiplier with its MSB off of pin $\alpha + k$ takes $t_{\alpha+k} = t_\alpha + k\delta$, where $\alpha, k \in \square$, $0 \le \alpha + k \le 35$, and δ is the slope of the line in Figure 10. The Multiplier Switching

Section in [18] shows the delay at each pin, from the LSB of the multiplicand to the MSB of the product. The synthesis reports show this linear relation (Appendix B.1).

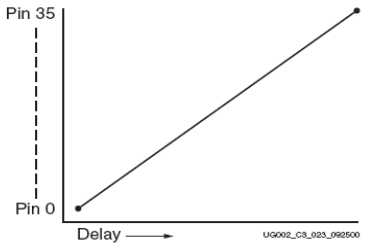


Figure 2-11: **Pin-to-Delay Ratio Curve**

Figure 10 Pin-to-Delay ratio curve for MULT18x18. (From [19])

3. BRAMs

BRAMs are an integral resource on the Virtex-II. They are arranged in columns between the MULT18x18s and the CLBs. This reduces the delay between memory and the multipliers. Each of the 6 columns contains 24 BRAMs. Each BRAM contains up to 18Kbits, and can be configured in various word widths, (1 to 36 bits). Thus, each BRAM uses 9 to 14 address lines, depending on the width of the word stored. There are a total of 324Kbytes of data storage in BRAMs on the Virtex-II XC2V6000.

B. SOFTWARE

This section discusses the software that was used to obtain simulation data and to estimate complexity and delay for NFG components.

1. Xilinx ISE Project Navigator

Xilinx ISE Project Navigator was used extensively for designing, simulating and synthesizing various arithmetic logic devices. The software suite includes schematic and VHDL editors along with a library of hardware primitive components. In some cases, behavioral VHDL modules were created, and, in other cases, schematic modules were created. Once a particular module was created, it was synthesized to provide estimations

of hardware utilization and worst case propagation delays. Examples of the synthesis reports are contained in Appendix B.1.

2. MATLAB

MATLAB was also used extensively. MATLAB was used to plot data obtained from the synthesis reports. It was also used to import the same data and to estimate hardware utilization and delay for various arithmetic devices. It was used for visual analysis of NFG hardware utilization and propagation delays. A summary of the MATLAB source code is in Appendix A.

C. DATA COLLECTION AND ESTIMATION

In order to analyze a particular NFG's hardware utilization and propagation delay, it is necessary to have data on the particular arithmetic components that are used by the NFG. For example, if an NFG requires a 23x23-bit multiplier and a 46-bit adder, then it is necessary to know the hardware utilization and propagation delay for the 23x23-bit multiplier and the 46-bit adder. The goal of collecting the data for this thesis is to obtain relatively accurate measurements in order to be able to estimate complexity and delay parameters without having to implement a specific logic design of each NFG. In addition, it might be required that we compare this same NFG to a similar one with a 22x22-bit multiplier and a 44-bit adder. Since it is impractical to construct multipliers, adders (and other arithmetic devices) of every possible size, only a subset of sizes were considered. The pertinent information was gathered from the synthesis reports into the text files in Appendix B.2. Timing data from the synthesis reports was used because it was accurate to 1ps. Timing information provided in [18] was only accurate to 10ps, but still confirmed the data obtained through simulation. Since the data did not cover all possible sizes, estimates were made so that a data point exists for components of all sizes ranging from 1-bit components up to 129-bit components. In some cases, such as the Ripple Carry Adder (RCA), equations were developed that match all of the simulation data points. In other cases, such as the multiplier, missing data points were estimated

using linear approximations. Device architectures and trend analysis of the data points were both considered when deciding what data points to collect.

1. Making Linear Approximations for Missing Data Points

The author's MATLAB function fillLin takes scattered ***x*** and ***y*** data points, given in array form, and estimates the data points in between the given ***x*** values. The array ***x*** must be an array of monotonic increasing integers. The length of the array ***x*** must be the same as the array ***y*.** This is applicable to this thesis because this function will estimate a parameter of an *n*-bit sized device, where $n \in \square$. The array ***x*** holds the *n* values in the collected data tables, and the array ***y*** holds the propagation delay values or the hardware utilization values. The fillLin function produces an array ***y'*** where the index ranges from 1 to the maximum value of the original ***x*** array, and the value is the estimated function value evaluated at the index number. For example, to approximate a known function $f(x) = x^2$, where data points are taken at ***x*** = 1, 2, 4, 7, and 9, call the function in MATLAB with the array ***x*** = [1 2 4 7 9], and the corresponding array ***y*** = [1 4 16 49 81]. The function "fillLin" returns the array ***y'*** = [1 4 10 16 27 38 49 65 81]. The array ***y'*** is now 9 elements long, and has a value for every integer *x*, ranging from 1 to 9. To obtain *y*(3), or 3^2, simply call ***y'*** with 3 as the index into ***y'***, resulting in *y'*(3) = 10. Of course, this example illustrates the inaccuracies of the approximation, but as more data points are collected, better approximations occur. Also, this function is applied only to monotonically increasing functions, namely hardware utilization with respect to word size, and propagation delay with respect to word size. As the word size of an arithmetic device gets larger, both complexity and delay get larger. Even slightly inaccurate estimations still provide a value that can be used for general comparisons. Figure 11 shows how fillLin fills in the missing data from collected data points with linear approximations to form a continuous function where the input is an integer from 1 to at most 129.

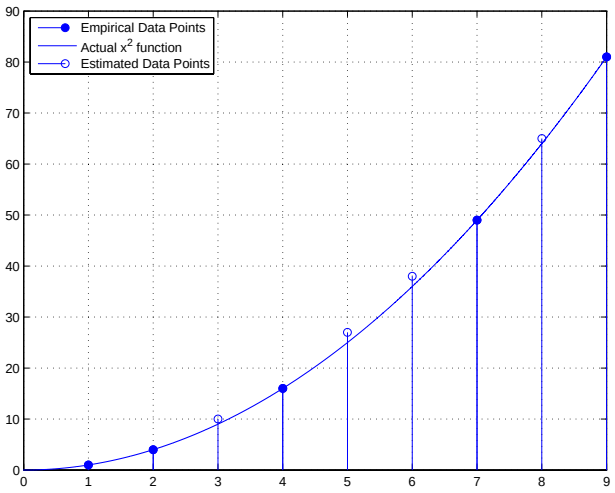


Figure 11 Example of fillLin Approximation for $y = x^2$.

The graph in Figure 12 shows the application of the function fillLin to the data collected for the net delays. The stems represent the actual data points collected. This means that propagation delays were collected for several fanouts. If a designer needs to find the net delay for a particular node with a fanout of 100, it is easy to extract that information from the array created by fillLin. Data collected for several components is shown in Appendix B.2. The graphs of fillLin results for these data points are shown in Appendix B.3.

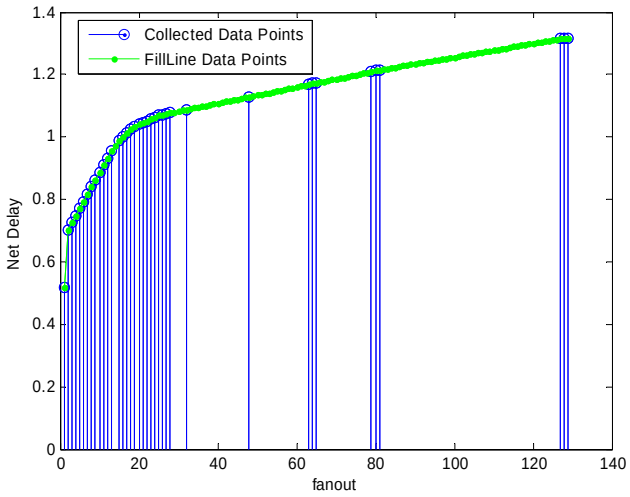


Figure 12 fillLin Function (Using Data Points from Net Delay).

The fillLin function yields an accurate representation without having to collect data points to fill the entire x-axis. The accuracy of the fillLin function is not analyzed in depth in this thesis because the errors in estimation are relatively minute. For example, a visual inspection of Figure 12 shows that when the largest jump between data points occurs between fanout values of 81 and 127. The approximate distance in net delay between the two fanouts is 0.1 ns. Assuming basic knowledge of net delay vs. fanout, we can say that net delay is monotone increasing between successive data points. Therefore, the maximum error possible for fanout is 0.1 ns, which is relatively minute. The actual error is most likely much smaller than 0.1 ns. However, when fewer data points are collected, the relative errors can be large. To minimize these errors, specific data points are collected based on analysis of component architectures.

When collecting data to enter into the function, data points were collected at key positions so that a piecewise linear approximation of the complexity and delay equations would be accurate. It was verified that midpoints corresponded to projected linear approximations. The purpose of the steps above is to develop a function that returns the delay or complexity of a circuit element based on the number of input bits, and the type of element. For example, if an NFG requires a 23x23 bit multiplier, the function returns an accurate time delay without building and synthesizing it; its complexity and delay are computed by interpolating between a value of n above and below n=23.

In some cases, it was possible to determine an actual function from the data points. For example, the delay of an RCA versus word-size is a linear function for $n > 4$. For these instances, the linear equation is used to approximate time delays and/or complexity, and 'if' statements replace the delay value for data points that don't fit the approximation equation. In the case of the RCA, for n=1 to 4, a simpler architecture is possible, so specific data points are used to give the delay and size estimates. In general, all devices exhibit nonlinear behavior of delay and size versus n when n is small because there are multiple ways route signals inside each slice of the FPGA. Each of these signal paths have different delays based on the particular electronic device through which it is routed.

Data was collected at various word-widths *n* for net delays, which are based on an *n*-bit fanout, *n*x*n*-bit unsigned multipliers, *n*-bit RCAs, *n*:1 MUXs, *n*-address bit distributed RAM/*n*-input functions/*n*-address bit ROM and BRAMs. Other devices can be constructed from these basic elements.

2. Measuring Hardware Complexity

It is difficult to measure hardware utilization when there are different types of resources, each having a different quantity. This section describes the how each resource is consumed, and how a single measure can be used to describe overall hardware utilization based on the utilization of each resource.

a. Deciding on the Basic Units of Measurement

Since there are multiple ways to organize the basic signal flow through a CLB, it is complicated to find a common method to quantify how much space a circuit takes up. In some instances, a device might use only 1 LUT, but also use multiple MUXs in the same slice. Thus, even when only 1 LUT is used, it may still prevent the use of the rest of the slice by other circuitry. The synthesis reports from Xilinx ISE Project Manager include the number of slices used, α, and the number of LUTs used, β. However, α may be more than 2β, suggesting that not all of the slices use both of its LUTs. For this reason, we measure hardware utilization in terms of slices utilized. Doing so puts everything in common terms that are verifiable with the software being used.

Likewise, the synthesis reports include the number of MULT18x18s and BRAMs used in a particular design. No partial resources are used. Even if only 2 bits of a MULT18x18 are used, it consumes the entire resource. If only 2-bytes of RAM are implemented in a BRAM, then it consumes the entire block of memory. Thus, the basic unit for measurement of MULT18x18s is 1 MULT18x18, and the basic unit for BRAMs is 1 BRAM.

b. Finding Meaningful Terminology for Measuring Hardware Utilization

Since three resources are considered, there are three terms for hardware utilization. The *slice utilization percentage* (SUP) is defined as percentage of the slices that are required in order to implement a specific logic circuit design, based on the data from the synthesis reports (see Appendix B.1). Likewise, the *multiplier utilization percentage* (MUP) and *BRAM utilization percentage* (BUP) are defined as the percentages of respective resources used to implement a specific circuit design. Table 2 summarizes the equations for calculating these measures, using the quantities of resources given in Table 1.

$\text{SUP}=\frac{\#\text{ slices utilized}}{\text{total \#slices on FPGA}}\times 100\% = \frac{\#\text{ slices utilized}}{33792}\times 100\%$
$\text{MUP}=\frac{\#\text{ MULT18x18s utilized}}{\text{total \#MULT18x18s on FPGA}}\times 100\% = \frac{\#\text{ MULT18x18s utilized}}{144}\times 100\%$
$\text{BUP}=\frac{\#\text{ BRAM utilized}}{\text{total \#BRAM on FPGA}}\times 100\% = \frac{\#\text{ BRAM utilized}}{144}\times 100\%$
$HUP = 100\% - \sqrt[3]{(100\% - SUP)(100\% - MUP)(100\% - BUP)}$

Table 2 Equations for SUP, MUP, BUP, and HUP.

It is often useful to compare devices that use more than one resource at a time. For example, large multipliers consume onboard MULT18x18s, but also require partial product adders which consume CLBs. Consider comparing an NFG that uses this multiplier with one that uses a large ROM instead. The ROM might consume only BRAMs. A SUP, MUP and BUP can be calculated and compared for each NFG, but there is no way to compare overall hardware utilization. For this reason, the *hardware utilization percentage* (HUP) is computed as a function of the SUP, MUP and BUP. The function shown in Table 2 is used because it exhibits desirable characteristics. When any single resource is consumed (i.e. SUP, MUP, or BUP $\geq 100\%$), HUP = 100%, indicating

that the required resources are not available on the Xilinx Virtex-II XC2V6000 FPGA. This does not necessarily mean that the NFG cannot be implemented on this particular FPGA. It means that the models developed in this thesis no longer provide accurate estimations for HUP and delay. Each model assumes that particular components are used. For example, if an NFG requires 169 MULT18x18s, it could be possible to implement it on a single FPGA by building the additional 25 multipliers from CLBs. However, the models do not take this into account. Thus, when the HUP for a particular NFG reaches 100%, it shows that the models will not be able to accurately represent complexity and delay for larger NFG sizes.

When a particular logic device uses all three resources proportionally (i.e. SUP=MUP=BUP), then the HUP function behaves linearly. When only one resource is consumed the HUP function behaves like a cubed-root function. The cubed root function still offers a meaningful relation between hardware utilizations of NFGs that use different resources. As more hardware is used, the HUP increases. The HUP increases slightly less than it would if all resources are consumed proportionally. Figure 13 shows an example where the hardware resources are used proportionally (i.e. MUP=SUP=BUP), where slices are used without any other resources (BUP=MUP=0), and where slices and MULT18x18s are used proportionally but without any BRAMs (MUP=SUP, BUP=0).

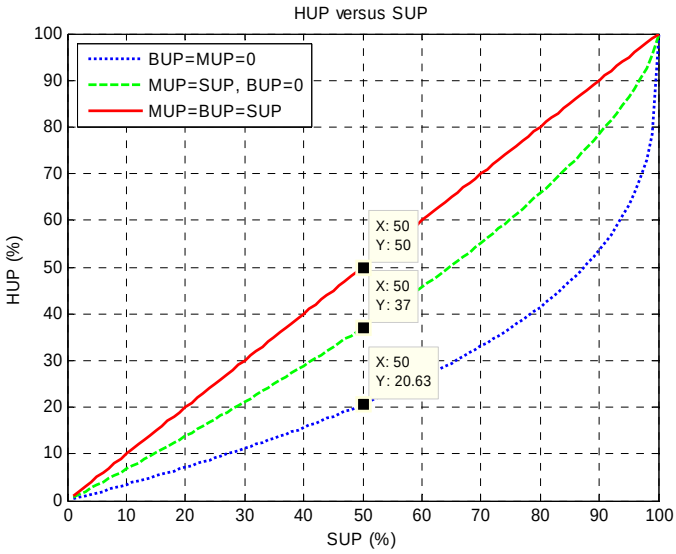


Figure 13 HUP vs. SUP for Various BUPs and MUPs.

Since the variables SUP, MUP and BUP are weighted evenly within the HUP equation, the same relationships apply when a single resource is used, regardless of what resource is used. In general, arithmetic components do not consume all three resources proportionally. Multipliers consume MULT18x18s and CLBs in uneven proportions, and coefficient tables consume BRAMs and CLBs in uneven proportions. The majority of the arithmetic components analyzed in this thesis consume only one type of resource. Figure 14 shows another example where the BUP=SUP for various MUP. When the MUP=0, the HUP curve shows two resources being consumed proportionally. When MUP=50%, note that the HUP begins at approximately 20%. Thus, when 50% of the MULT18x18s are used, it is considered that at least 20% of the total FPGA resources are used. When 95% of the MULT18x18s are used, at least 90% of the total hardware is used.

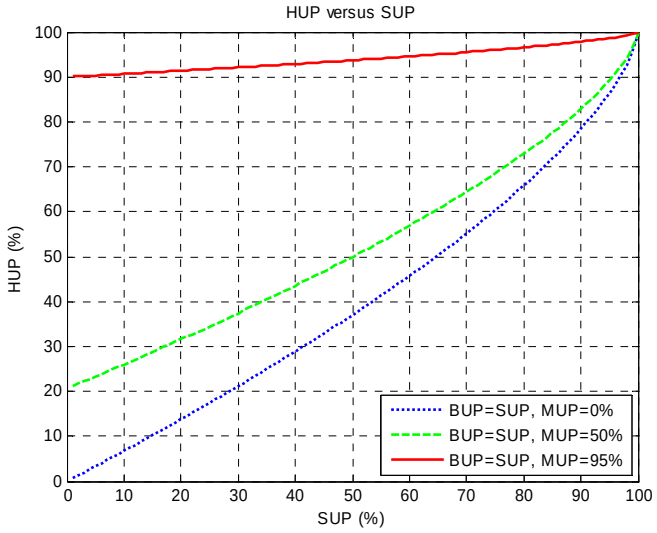


Figure 14 HUP versus SUP where BUP=SUP for various MUPs.

It should be noted that the HUP equation in Table 2 does not exhibit desirable properties when SUP, MUP, or BUP are greater than 100%, thus the MATLAB function HUP.m caps each at 100%. This produces a maximum HUP of 100%. When HUP = 100%, it indicates that the complexity and delay of the NFG being analyzed is not accurate because there are not enough of at least one of the resources that it needs.

3. Measuring Propagation Delay

The goal of this section is to determine how to accurately measure the propagation delay of a given circuit, without having to build that particular circuit and simulate it. Signal propagation delay depends on the path over which the signal propagates. Thus, the general architecture of the circuit must be understood in order to know what delays are encountered by a given signal. In this section, we are concerned with finding the longest propagation delay for each particular circuit. In cases where architectures are simple, such as the adder (section E.1), accurate expressions are straightforward. For other cases, such as the multiplier (section E.2), data is collected from simulation results and estimates are made to represent missing data. In both cases, it is important to understand the source of the delays. Timing data was acquired using a low-level synthesis tool in Xilinx ISE Project Navigator. In some cases, it was simple to correlate the timing data from the synthesis reports to the data supplied in [18]. In other cases, timing data from the synthesis reports alone was used. The following delays are discussed to better understand their contribution to propagation delay.

a. Net Delay

Net delay (t_{net}) is common to all circuits designed on FPGAs. Net delay is a propagation delay that is due to transferring charge along a wire. It is proportional to the size of the wire or conductor and inversely proportional to drive strength of the associated power supply. DC power supplies can only supply a limited amount of current. On an FPGA, the drive strength for a given node is dependent on what driver, such as a logic gate, register or IOB, is connected to the node. The time it takes to charge a given wire to a desired voltage is also dependent on the fanout of the driver. If the driver supplies charge to more inputs, then more charge is required, resulting in a longer time delay for the entire wire to build to the required voltage. Figure 15 shows an example of a schematic circuit built in Xilinx Project Navigator to collect net delay data for various fanouts. Appendix B.2 contains the data collected for net delays.

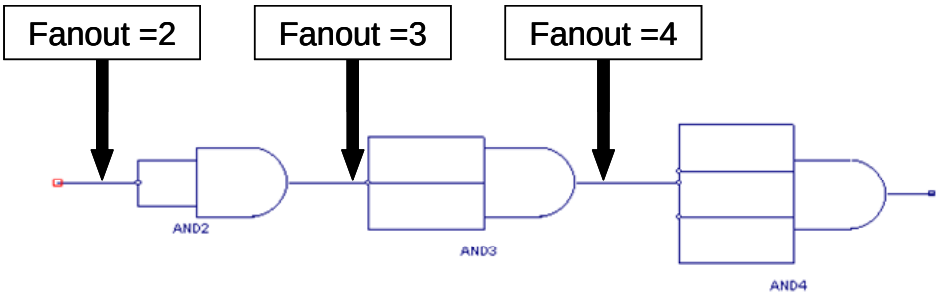


Figure 15 Schematic Example of Various Fanouts.

When designing arithmetic logic devices, the net delay is significant because some architecture have relatively large fanouts. Net delays on the Xilinx Virtex-II XC2V6000 FPGA with speed grade of -4 ranges from 0.517 ns to 1.316 ns based on synthesis reports for various circuits. Figure 16 shows the net delay versus fanout that is generated by the function fillLin when given the collected data as an input. Although the net delay is generally smaller than the delay of logic components, when multiple logic stages with high fanouts are cascaded, the associated net delays can be a significant contribution to the total combinational delay of the circuit.

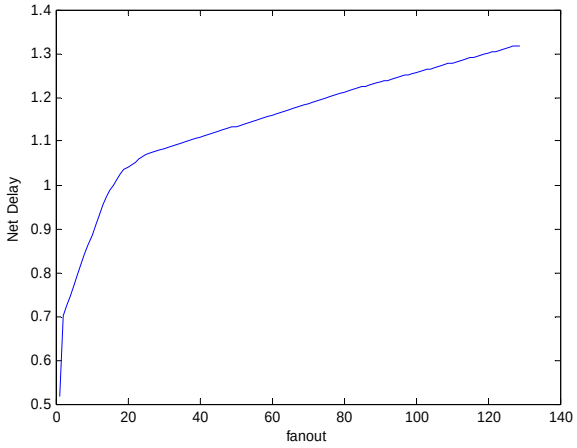


Figure 16 Net Delay vs. Fanout after fillLin.

When estimating propagation delays for various arithmetic devices (see Section E in this chapter), the file HUandDelay includes the net delay going into the particular device. However, it excludes the net delay associated with the output because it

depends on the number of inputs driven by the output. This simplifies the calculation of propagation delays for composite circuits. Figure 17 illustrates the propagation delays associated with combining two arithmetic devices in series. The total propagation time through the composite circuit is $t_{prop} = t_{net,1} + t_{comb,1} + t_{net,4} + t_{comb,2}$, where $t_{net,\kappa}$ is the net delay associated with a fanout of κ, and $t_{comb,j}$ is the combinational delay of the j-th arithmetic device in series.

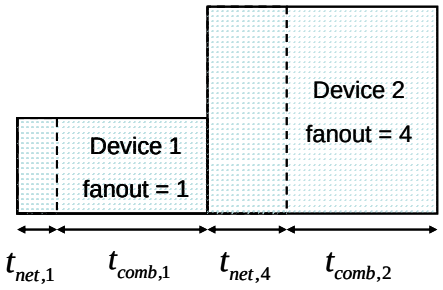


Figure 17 Propagation Delay for Arithmetic Devices in Series.

When arithmetic devices are placed in parallel, the fanout of the input wires becomes the sum of the fanouts of each device, and the net delay for each device requires adjustment. If not, small errors (up to 0.8 ns) are introduced in propagation delay estimations every time devices are placed in parallel. In most NFGs, this error is insignificant, but this thesis uses the correct net delays. Figure 18 illustrates how this error affects the propagation delay of the composite circuit.

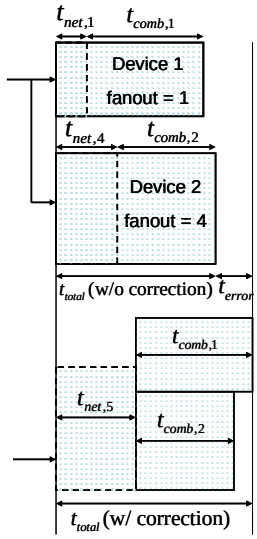


Figure 18 Propagation Delay for Arithmetic Devices in Parallel.

b. LUT Delays

LUT delays are the propagation delays associated with a signal propagating from the input of LUT (or function generator) to the output of the LUT. LUT delays are denoted as t_{LUTq}, where $q \in$ and $1<q<6$. [18] reports t_{LUT4} to be 0.44ns, and synthesis reports demonstrate this value to be 0.439ns. The delay is the same for LUTs even if all four inputs are not used. Thus, $t_{LUT1} = t_{LUT2} = t_{LUT3} = t_{LUT4}$. Five-input LUTs can be formed using two 4-input LUTs and a specialized MUX within the same slice. According to [18], $t_{LUT5} = 0.72ns$. The additional delay is due to the MUX that is needed to combine two 4-input LUTs to form a 5-input LUT.

c. Delays in Special Purpose MUXs

As discussed previously, there are various MUXs in each slice that can be configured for use in design of a logic circuit. This section identifies some of the propagation delays associated with the MUXs that are used in the arithmetic devices in this thesis.

MUXCY, shown in Figure 8 provides a path for fast carry logic used to implement an adder. The two delays of concern are $t_{MUXCY,S\rightarrow O}$ and $t_{MUXCY,I0\rightarrow O}$. The first delay, $t_{MUXCY,S\rightarrow O}$, is the time it takes to change the output *O*, after the select line *S* changes. The second, $t_{MUXCY,I0\rightarrow O}$, is the propagation delay of a signal from input *I0* to the output *O*. Empirical evidence from Xilinx ISE Project Navigator confirms data from [18] for the values in Table 3

Parameter	Time delay (ns)
$t_{MUXCY,I0\rightarrow O}$	0.053
$t_{MUXCY,S\rightarrow O}$	0.298

Table 3 MUXCY Propagation Delays.

MUXFX is designed to combine signals from multiple slices into a single output. This is useful when constructing functions of more than 4 variables. For example, instead of cascading multiple layers of 2:1 MUXs built from LUTs, larger MUXs are constructed from the built-in MUXFXs. This eliminates the net delays associated with interconnecting LUTs. For example, a 4-input function takes 0.44ns plus a net delay to produce a result, while a 5-input function takes only 0.72ns and a net delay (vice $2x0.44ns = 0.88ns$ and two net delays for two cascaded LUTs).

d. IOB Delay

Timing data was acquired using a low-level synthesis tool in Xilinx ISE Project Navigator. The synthesis includes estimated routing delays (net delays), combinational delays, and Input/Output Buffer (IOB) delay. Since NFGs would most likely cascade multiple arithmetic and/or memory units together, IOB delay data is removed from the total delay for the particular component. For example, the total delay of an NFG that is comprised of a RAM unit propagating into a multiplier, then into an adder, is the sum of the combinational delays of each component and the estimated routing delays. The low level synthesis provides timing data along the longest combinational path, and includes the IOB delays, net delays, and combinational delays based on the routing through each slice. The data collected in LoadISEDeviceData removes the IOB delays and contains the net delays.

D. ESTIMATING PARAMETERS FOR VARIOUS BASIC ARITHMETIC LOGIC COMPONENTS

Various NFG configurations require various arithmetic logic devices in series and/or in parallel. This section discusses measuring the complexities and propagation delays for common arithmetic logic devices applicable to NFGs. It describes simple architectural designs for several circuits, which are not necessarily the most efficient or compact designs. The goal is not to find the best case hardware design, but to use commonly accepted methods to build basic arithmetic circuits in order to compare

complexities and propagation delays. The measurements of the arithmetic circuits in this section are used to measure the overall complexity and delays for the NFG configurations that are built from them.

The author's MATLAB function HUandDelay.m calculates the SUPs, MUPs, BUPs, and delays of several components. These parameters are calculated based on the particular component having *n* input bits and *w* output bits. The number of output bits is only used for memory components and SIEs. Table 4 summarizes each function handled by HUandDelay.

Inputs variables		**Output Variables :**
n,w	**Device Name**	**SUP, MUP, BUP and propagation delay for a(n):**
n,w	'ROM' 'LUT'	*n*-input w-output function, or a single bit ROM with n address lines (2^n x *w* ROM).
n,w	'Adder'	adder with 2 input vectors of length *n* and a carry in bit, and a single output vector of length n, plus a carry out bit. Note: w is not used.
n,w	'Mult'	multiplier with 2 input vectors of length *n*, and a product vector of length 2*n* (built from CLBs only, no MULT18x18s are used)
n,w	'Mult18x18'	multiplier with 2 input vectors of length *n*, and a product vector of length 2*n* (built from CLBs and MULT18x18s)
n,w	'MUX'	*n*:1 MUX, with $n+\lceil \log_2 n \rceil$ input bits, and 1 output bit
n,w	'BarrelShifter'	*n*-bit barrel shifter with $n+\lceil \log_2 n \rceil$ input bits and *n* output bits
n,w	'BRAM'	memory unit constructed from onboard BRAM units, with *n* address bits in, and w-bits out (2^n x *w* RAM).
n,w	'SIE'	Segment index encoder with *n* input bits, and *w* output bits.
n,w	'SOP'	worst case Sum of Products logic circuit with *n* inputs and w output bits
n,w	'Mem'	best case memory unit constructed from BRAMs or from ROMs, 2^n x *w* ROM

Table 4 Summary of "HUandDelay" Operations.

The component designs do not necessarily represent the best case design or the worst case design. They are merely working designs that have been constructed from either behavioral VHDL models or from schematic models that can be implemented efficiently onboard the FPGA. Bit widths up to 129 bits wide are analyzed.

1. Adders and Subtractors

Adders and subtractors are commonly used arithmetic logic devices. Since a subtractor can be constructed with almost equivalent complexity to an adder, only the adder architecture is analyzed. For NFGs that require subtractors, adders are substituted because they exhibit the same characteristics.

a. Architecture

Xilinx FPGA architecture has been specifically designed for fast mathematic operations, including additions and multiplications. Fast carry chains are built in columns that run through each slice via fast MUXs, namely the MUXCY (see Figure 8). The propagation delay from one bit to the next is approximately 53 ps. Even large RCAs can compute a large number of bits relatively quickly. Each fast carry chain can be 176 bits long [18]. This means that the carry propagation portion of the adder's delay is only 9.3 ns for a 176-bit adder. Longer carry chains can be constructed by connecting the last carry out to another fast carry chain, but associated net delays are attached. However, an adder wider than 176-bits is not generally required in NFGs. Contrary to conventional logic design, using Carry Look-Ahead (CLAH) architecture actually produces slower adders due to the additional XOR logic depth. Figure 19 shows how a single bit full adder is implemented using a LUT, MUXCY, and XOR within half of a slice. Note that each LUT is configured as a two-input XOR gate, having the same delay as a 2-input LUT, t_{LUT2}.

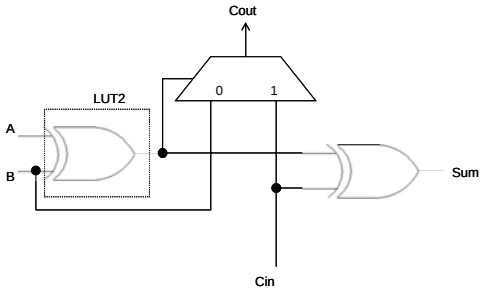


Figure 19 Single-bit Full Adder Implemented on Virtex-II FPGA.

b. Complexity Analysis

The goal of this complexity analysis is to find an accurate method to quantify hardware utilization for adders based on the size of the adder. Figure 20 shows the logic and carry path of an *n*-bit full chain implemented in $\left\lceil \frac{n}{2} \right\rceil$ slices. Thus, an *n*-bit adder occupies $\left\lceil \frac{n}{2} \right\rceil$ slices. Empirical data in the synthesis reports also confirms this relationship. The number of slices is calculated using the ceiling function in the author's function HUandDelay (Appendix A.2) and is used to find the SUP (Table 2). Because adders do not use MULT18x18s or BRAMs, the function returns MUP=0 and BUP=0 for an *n*-bit adder.

c. Delay Analysis

The propagation delay of an RCA is linear. Behavioral models for adders implement RCAs on the Virtex-II, so the propagation delay is expected to be linear. Data collected from the synthesis reports confirm this for $n > 4$. The data used for propagation delays does not include IOB delays, but does include net delays. By tracing the propagation path given by the synthesis reports, as shown in Figure 20, the total delay is derived to be $t_{prop} = t_{MUXCY,I0 \to O} \cdot (n-2) + t_{overhead}$, where $t_{MUXCY,I0 \to O} = 0.053ns$ and $t_{overhead} = 2.528ns$. According to [18], the carry delay through the fast MUXCY from input *I0* to output *O* is 0.05 ns, correlating the theoretical expectation and empirical data to the manufacturer's specifications. Also, there is no carry propagation in a single-bit or

a 2-bit adder since they are within the same slice. For larger RCAs, the first and last MUXCY do not lie in the longest propagation path. Thus, the delay along the carry propagation path is proportional to *n*-2, and the overhead delay accounts for the rest of the time delay through the RCA. Figure 20 shows the total propagation delay path through a RCA implemented on the Xilinx Virtex-II XC2V6000 FPGA.

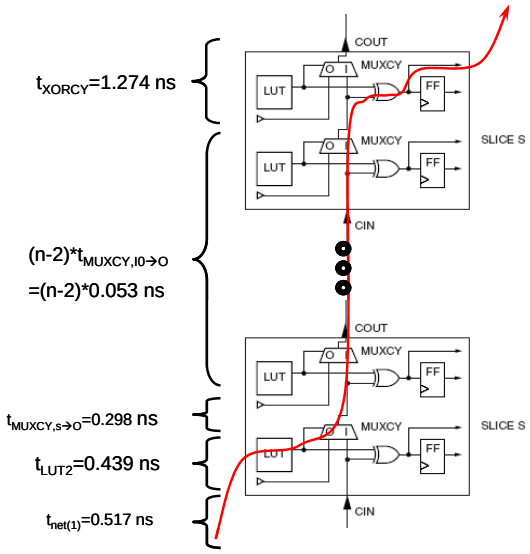


Figure 20 An *n*-bit RCA Propagation Delay Path on Xilinx Virtex-II. (After [18]).

The remaining portion of the equation can be verified by breaking down $t_{overhead}$ into the delays of the other logic components with the slices containing the LSB and the MSB. Switching characteristics for these components (in [18]) correspond to the signal path delays found in the synthesis reports. For example, $t_{overhead} = t_{XORCY} + t_{net(1)} + t_{LUT2,IO\rightarrow O} + t_{MUXCY,S\rightarrow O}$, where $t_{XORCY} = 1.274ns$, $t_{net(1)} = 0.517ns$, $t_{LUT2,IO\rightarrow O} = 0.439ns$, and $t_{MUXCY,S\rightarrow O} = 0.298ns$. The explanation of these terms can be found in [19], but are illustrated in Figure 20. The synthesis reports in Appendix B.1 show the delay of each component in the overhead common to all *n*-bit RCAs. Since a linear equation can accurately (to within 0.01ns) represent the simulation data, the author's MATLAB function HUandDelay returns the propagation delay of an *n*-bit RCA by calculating it with the same linear equation instead of using a table of referenced values. This allows a simple calculation to accurately return a valid

propagation delay for a given RCA size *n*. Figure 21 shows the overall SUP and propagation delay for adders versus the size of the adder.

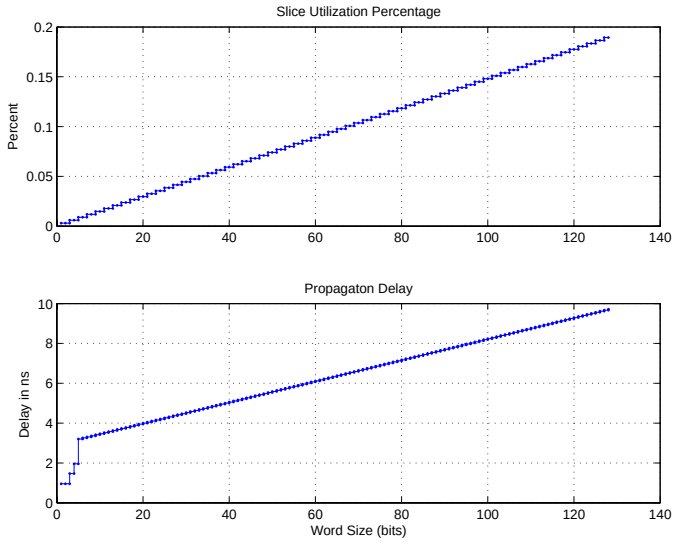


Figure 21 SUP and Propagation Delay for *n*-bit RCAs.

2. Multipliers

In order to understand hardware utilization and propagation delays for multipliers, it is necessary to understand their architecture

a. Architecture

Array multipliers generally require partial product generators (PPGs) and PP adders. Figure 22 shows the general architecture of an n-bit multiplier using PPGs and RCAs. The hardware utilization percentage (HUP) and propagation delay of an *n*x*n*-bit array multiplier depend on the number of PP multipliers required and the number of PPs that need to be added together. Relatively large multipliers may need to be analyzed for some of the applications in this paper. Xilinx's Virtex-II XC2V6000 FPGA includes 144 18x18-bit signed multipliers. Each one can be used as an *n*x*n*-bit multiplier for n< 18, or as a PPG for larger multipliers.

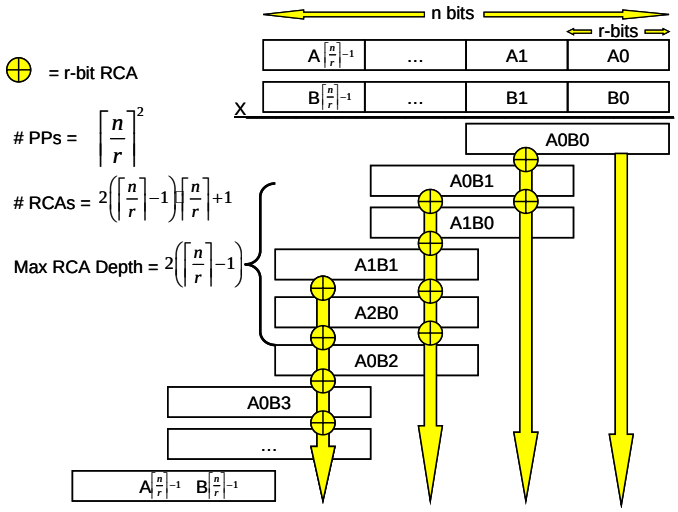
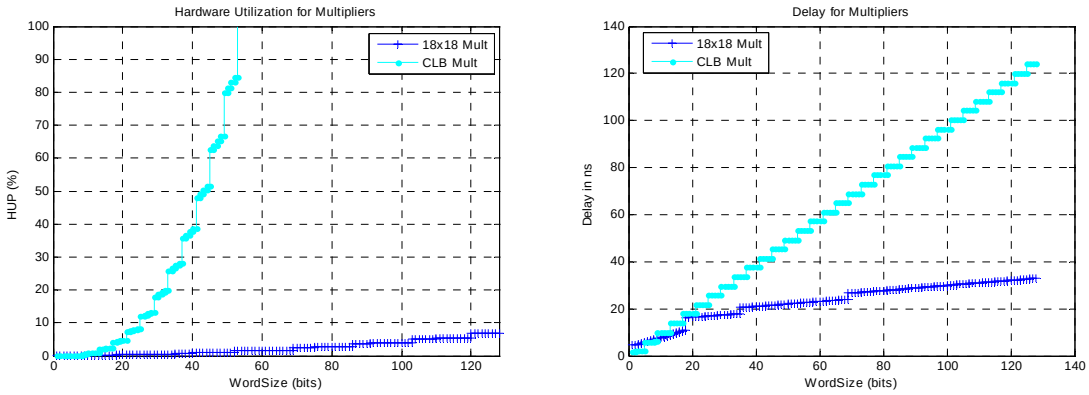


Figure 22 General *n*x*n* Array Multiplier Architecture.

The size of an array multiplier depends on the number of bits being multiplied. It also varies depending on the size of the PPGs. The most basic PPG is the 1x1 bit multiplier, which is an AND gate. A 2x2 bit multiplier is a 4-input to 1-output function, which can be realized in four LUTs. Since the number of function inputs grows proportional to n^2, the multiplier becomes very complex for larger PPGs if LUTs are used to realize the function. An *n*x*n*-bit multiplier designed with the architecture in Figure 22, requires $\left\lceil \frac{n}{r} \right\rceil^2$ PPGs and $2\left(\left\lceil \frac{n}{r} \right\rceil - 1\right)\left(\left\lceil \frac{n}{r} \right\rceil\right) + 1$ *r*-bit adders. Figure 23 illustrates the proportionality of multipliers' SUPs to $\left\lceil \frac{n}{r} \right\rceil^2$. As *r* gets smaller, more adders are required to sum the partial products. Figure 23 shows the HUP and propagation delays for a multiplier with *r*=4 built from 4-bit PPGs and 4-bit RCAs. It compares them to multipliers built using the MULT18x18s. Using MULT18x18s reduces the SUP for a multiplier, but also increases the MUP (see Table 2).



(a) HUP (b) Delay

Figure 23 Multiplier HUP and Delay vs. Multiplicand Size for Multipliers Built with MULT18x18s vs. CLBS.

Figure 23 shows that it is more efficient to develop large multipliers using the MULT18x18s on the Virtex-II FPGA. Here, the lower line represents multipliers built from MULT18x18s only, and the upper line represents multipliers built from LUTs only. Each one can be used as an *n*x*n*-bit multiplier for *n*< 18, or be an *r*-bit PPG for larger multipliers, where *r* <=17. Doing so takes advantage of the benefits discussed in section A.1.b. The propagation through each PPG is a linear function of $\left\lceil \frac{n}{r} \right\rceil$. For multipliers with *n*>17, all of the PPs can be calculated in parallel. This makes it more time-efficient to split *n*-bit multiplicands into $\left\lceil \frac{n}{r} \right\rceil$-bit multiplicands for each PPG, rather than using the maximum number of bits in a single MULT18x18 with fewer bits in the other required MULT18x18s. For example, if a 24x24 bit multiplier is required (Figure 24), it takes less time to compute four 12x12-bit multiplications in parallel using the MULT18x18s than it takes to compute one 17x17-bit multiplication in parallel with two 7x17-bit multiplications and a 7x7-bit multiplication (Figure 24). This is because the delay of the 17x17-bit multiplier takes longer than any of the other multiplications because the MSB of its product would come off of pin 34.

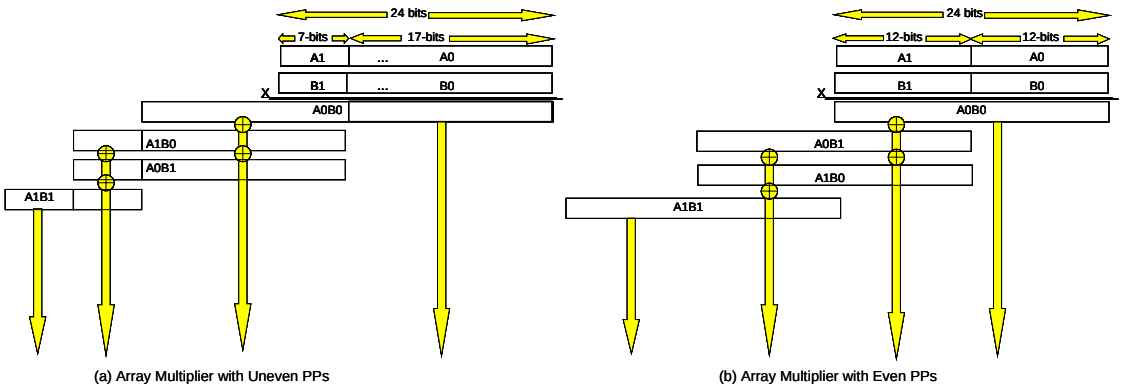


Figure 24 24-bit Multipliers with Uneven and Even PPs.

Since modern FPGAs incorporate multipliers, this analysis is usable for many other hardware applications as well. Array multipliers may be better designed using combinational logic. However, large multipliers require a larger portion of the CLBs on the FPGA and a much longer propagation delay. It is more efficient to use a few of the onboard MULT18x18s so that the CLB resources are available for other required logic devices. A 32x32 bit multiplier built from combinational logic consumes 24.9% of the slices on the FPGA and takes 29.9 ns to produce a result. The same multiplier built using MULT18x18s consumes only 0.14% of the slices and 2.8% of the MULT18x18s, and has a propagation delay of 17.7 ns. Since the objective is to establish a basic way to compare NFGs, and not to develop the most efficient *n*x*n*-bit multiplier, using the MULT18x18 onboard the FPGA as PPGs is a sufficient and reasonable method to build large multipliers, and results in a shorter propagation delay.

b. Complexity Analysis

Determining the size of a multiplier is much more complicated than the size of adders. For multipliers with $n<18$, a single MULT18x18 can be used, thus the percentage of MULT18x18s used, or MUP, is $1/144 \approx 0.7\%$. When more than one MULT18x18 is required, *r*-bit adders are required to sum the PPs. These *r*-bit wide PP adders consume CLBs. Therefore two parameters must be measured for any circuit

design that incorporates the on-chip MULT18x18s: MUPs and SUPs. If either the MUP or the SUP exceeds 100%, then the circuit being implemented will not fit on the FPGA. The HUP is shown in Figure 23.

Because array multipliers can be very complex, and can be constructed in various ways, it is not feasible, nor necessary, to dive deep into the architecture to analyze complexity in terms of CLBs. The adder is described in such a way that the architecture and product specification validated simulation results from the synthesis reports. Since simulation data was proven accurate for adders, it is assumed accurate for multipliers. Thus behavioral models of unsigned multipliers were designed and synthesized using ISE Project Navigator for various word widths. The synthesis reports provide the number of slices and MULT18x18s required, validating the quantity $\left\lceil \frac{n}{r} \right\rceil^2$ estimated in the architectural analysis above. These values are included in Appendix B.2, and are imported by HUandDelay to estimate hardware utilization using the linear approximation function fillLin.

c. Delay Analysis

For small multipliers, where $n<18$, the propagation delay is that of a single MULT18x18. Larger multipliers require multiple adders or adder trees. Again, the design of the multiplier can vary widely, which affects the delay. So to provide a simple method to provide relevant data, timing data is collected from the synthesis reports for the behavioral models. The propagation delays for various multiplier sizes are provided in Appendix B.2 and displayed in the graph in Figure 23.

3. Multiplexers (MUXs)

The NFG models in this thesis do not use MUXs. However, they are analyzed here so that future models can incorporate them. MUXs often perform vital functions (such as data signal routing) in arithmetic logic devices. For example, in a floating point systems [23], MUXs can be used to select an output from either a computed value or from a special number value (exact 0, NaN, $\pm\infty$) based on the whether or not the input is

a special number value. An *n*:1 MUX has *n* input bits, and routes only one of these inputs to the output bit depending on the bits used for selection. The number of selection bits required is $\lceil \log_2 n \rceil$. For example, a 16:1 MUX has 16 input bits (I0-I15) and 4 selection bits (S0-S3). To route input bit I7 to the output, the selection bits must be $0111_{2,}$ or 7_{10}.

a. Architecture

The Virtex-II architecture supports fast multiplexing by joining the LUTs within each CLB with MUXs built into each slice, thus minimizing propagation delays due to connecting to logic blocks in other CLBs. The delay is a nonlinear function with respect to size. By configuring each LUT to realize a 2:1 MUX, 1 slice can realize a 4:1 MUX by using the specialized MUXF5. Adjacent slices can be combined to form larger MUXs using the specialized MUXFX within each slice. Figure 25 illustrates the architecture of a 16:1 MUX built within a single CLB, or 4 slices. MUXs with *n*>16 can be built by combining multiple 16:1 MUXs with other MUXs.

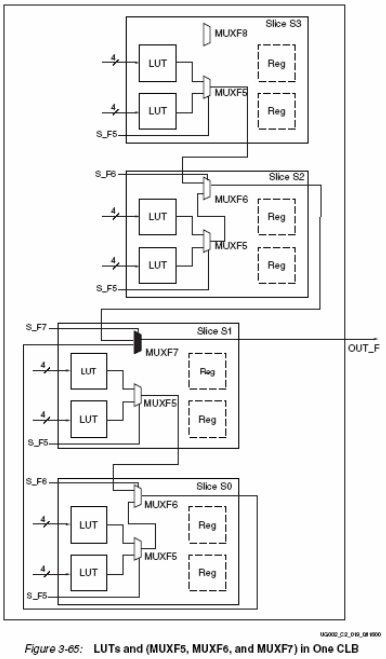


Figure 3-65: **LUTs and (MUXF5, MUXF6, and MUXF7) in One CLB**

Figure 25 16:1 MUX within a Single CLB. (From [18])

b. Complexity Analysis

Since four slices can implement a 16:1 MUX, the number of slices required in an *n*:1 MUX is $\left\lceil \frac{n}{4} \right\rceil$. To validate this approximation of hardware utilization based on the *n*, schematic models of MUXs where constructed in Xilinx ISE Project Manager. The schematics implement primitive MUXs included in Xilinx's library. The largest primitive MUX is a 16:1 MUX, which corresponds to the architecture described above. Larger MUXs were built by combining the primitive MUXs. For example, a 32:1 MUX was constructed by coupling two 16:1 primitive MUXs with a 2:1 primitive MUX. This method assures that an *n*:1 MUX is constructed in a compact manner. Synthesis reports for the schematic designs provided the data in Appendix B.2. The slice utilization data confirmed the estimates from the architectural description. The SUP for an *n*:1 MUX is calculated using the equation in Table 2. Since no MULT18x18s or BRAMs are used, MUP=0 and BUP =0 for all MUXs analyzed in this thesis.

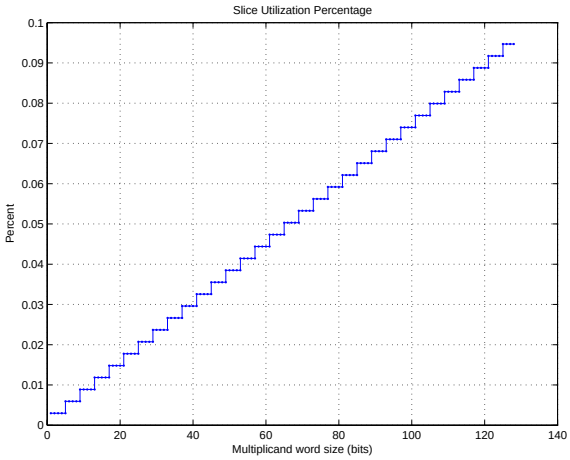


Figure 26 SUP vs. MUX size (bits).

c. Delay Analysis

The propagation delay through a large MUX depends on the number of MUX levels, and the delay through each particular MUX. Since different MUXs are used within each CLB, they each have a different propagation delay. The number of

MUX levels is $\lceil \log_2 n \rceil$. The synthesis reports provide propagation delay data for various MUX sizes. The data confirms the logarithmic relation between n and propagation delay.

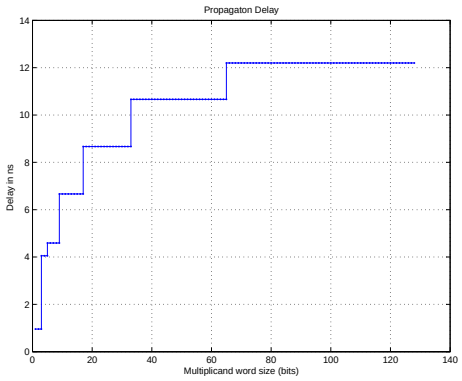


Figure 27 Propagation Delays vs. MUX Size (bits).

4. Barrel Shifters

Like MUXs, barrel shifters are not used in any of the models in this thesis. However, they are analyzed here because they may prove useful in reducing hardware complexity and delay for linear NFGs that restrict its slope coefficients (c_{1i}) to a power of 2. Barrel shifters can be used to realize multipliers when one of the multiplicands is a power of 2. They can be significantly faster and require fewer slices than a general multiplier. A basic *n*-bit barrel shifter consists of *n* *n*:1 MUXs in parallel. It shifts bits from the MSB into the LSB, or vice versa. A small amount of additional logic is needed to convert the basic barrel shifter into an arithmetic or logical combinational shifter.

a. Architecture

Figure 28 shows the general architecture of an *n*-bit barrel shifter, including the fanouts along the propagation paths. The darkened MUXs indicate that they can be considered a part of an *n*:1 MUX, multiplexing all inputs to a single output bit. The easiest method to build a barrel shifter would be to use *n* *n*:1 MUXs in parallel, one for each output bit. This is a naïve method since it does not reuse the 2:1 MUXs that can be reused. A better architecture is shown in Figure 28, containing $\log_2 n$ columns of *n* 2:1 MUXs.

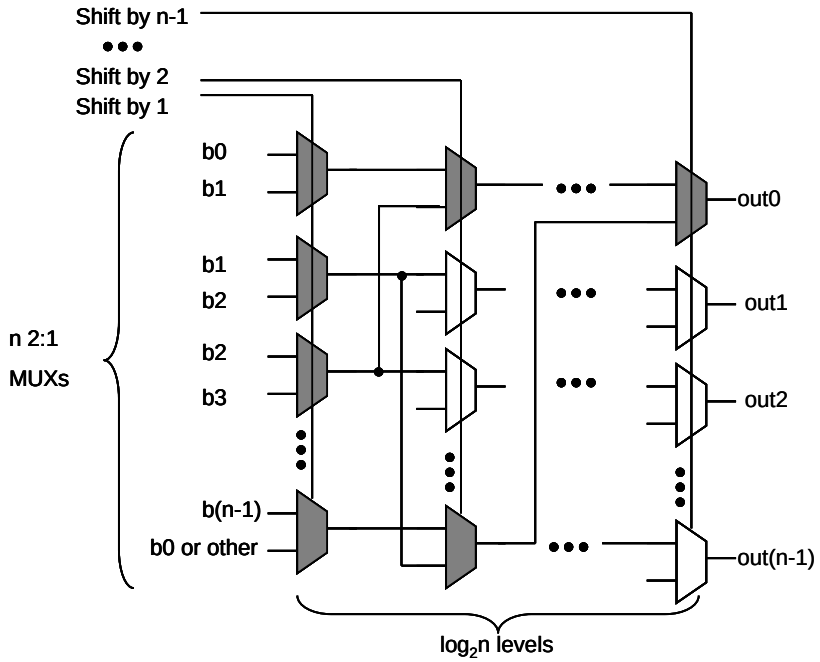


Figure 28 Barrel-shifter Architecture.

b. Complexity Analysis

An *n*-bit barrel shifter constructed in the naïve manner would consume n *n*:1 MUXs, or $\left\lceil \frac{n^2}{4} \right\rceil$ slices. The more hardware efficient method results in $\left\lceil \frac{n}{4} \right\rceil \log_2 n$ since each 2:1 MUX in Figure 28 can be constructed from a single LUT. The function HUandDelay uses the latter method.

c. Delay Analysis

The delay of an *n*-bit barrel shifter is closely related to the delay of an *n*:1 MUX. Because the shift-by-1 MUX select line must be distributed to all *n* 2:1 MUXs in the first column, the fanout of this line is *n*. Since the longest propagation path contains this select line, then a net delay based on that *n*, instead of 1, must be accounted for. Therefore, the barrel shifter's propagation delay is the same as an *n*:1 MUX plus the difference in net delays, or $t_{prop,BarrelShifter} = t_{prop,MUX} + t_{NET(n)} - t_{NET(1)}$. This equation is used in the function HUandDelay to return the propagation delay for an *n*-bit barrel shifter.

5. General Logic Functions

This section discusses various methods to implement general functions based on *n* inputs and a single output. These types of function may be used in NFGs as segment index encoders and relatively small coefficient tables.

a. Generic n-Input Functions

In the worst case, any *n*-input function can be realized with an *n*-input lookup table (LUT), which is functionally a ROM. The amount of required memory cells is 2^n per bit. Most functions can be reduced to smaller logic functions, so 2^n represents the upper bound of the required memory units. In Xilinx's Virtex-II FPGA, each LUT has 4-input bits, thus can represent a 4-input 1-output function or a 16x1 ROM. Thus, the number of LUTs required to realize any *n*-bit function or a 2^n x1 ROM is 2^{n-4}. Single-port RAM requires the same amount of LUTs, but can be read and written.

The delay of a 4-variable function realized by one LUT is 0.44ns [18]. The FPGAs are organized such that a 5-input function can be realized with in one slice, without having to cascade the delays, thus yielding a 0.72ns delay for a 5-input function. The overall delay through an *n*-bit ROM from an input to an output depends on whether the complete function is designed using cascades of 4-bit functions or 5-bit functions. Larger functions require combining 4 or 5-input functions with a MUX large enough to accommodate a total of *n* input bits. For the purpose of NFG comparisons, a ROM performs the same function and utilizes the same hardware as an *n*-input function. Thus, ROM primitives were constructed schematically in ISE Project Manager. The synthesis reports provided timing and hardware utilization data for ROM with up to 7-bit addresses. Larger ROMs are constructed from the largest ROM primitive. Thus, an *n*-address bit ROM requires 2^{n-7} 7-bit address ROMs and a 2^{n-7}:1 MUX. An example architecture is shown in Figure 29. HUandDelay imports timing data for the primitive ROMs from the data set in Appendix B.2, and recursively calls itself to find the additional hardware and propagation delay of the required MUX. The propagation delay takes into account the delays from the ROM, the MUX, and the net delay associated with

connecting the two devices together. Figure 30 shows the hardware utilization and propagation delay of an 2^n x1 ROM. Note that for *n*> 14, it is more efficient to use BRAM for implementing a large LUT instead of consuming a large number of slices. HUandDelay automatically selects BRAM implementation for LUTs with *n* larger than 14.

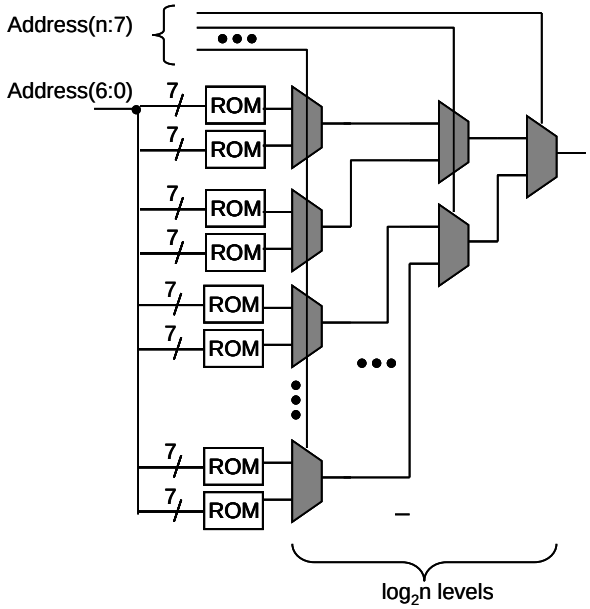


Figure 29 An *n*-input Function Using 7-bit Address ROMs and a 2^{n-7}:1 MUX.

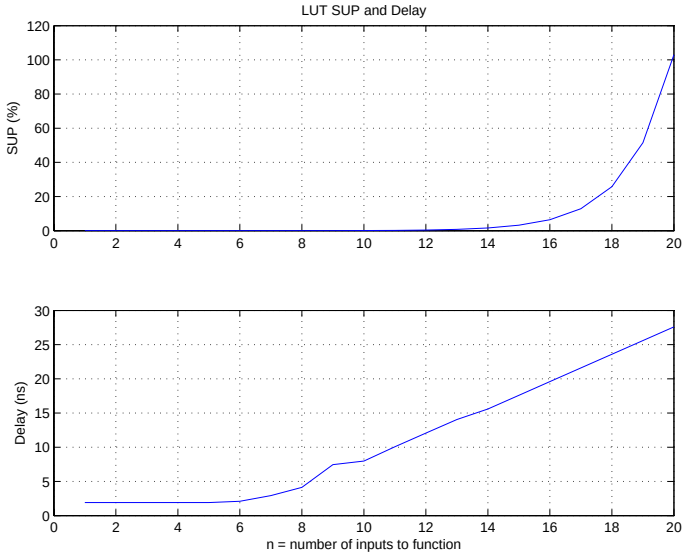


Figure 30 LUT SUP and Delay vs. Number of Address Bits for a ROM.

b. Sum of Products (SOP) Functions

A sum-of-product is a logic function of the form $\sum_{i=1}^{p}\left(\prod_{j=1}^{q_i} g_i\right)$ where p is the number of terms, q is the number of inputs into a term, and g is each input bit. Significant hardware and propagation delay reductions can be realized when a particular n-input function can be represented in a SOP form. The Virtex-II architecture is designed to efficiently implement wide SOPs. It is a difficult problem to determine the complexity of SOPs for logic functions. Benchmark functions tend to have small SOPs [22].

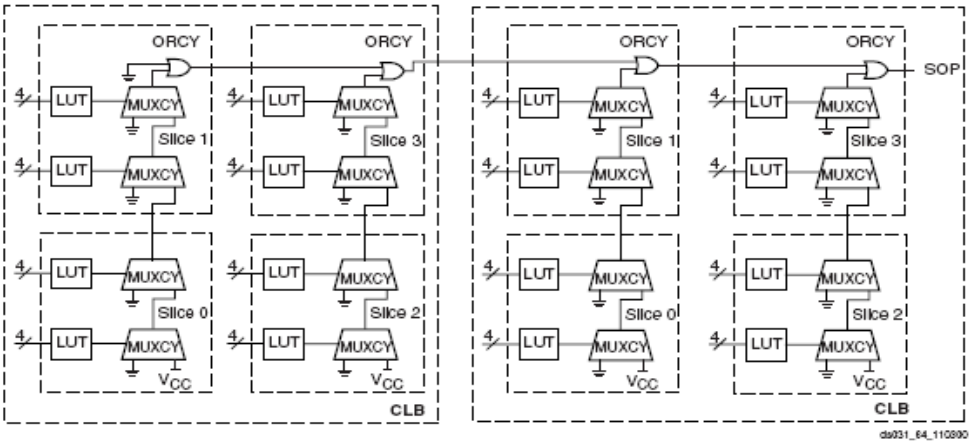


Figure 25: **Horizontal Cascade Chain**

Figure 31 SOP implemented on Virtex-II. (From [18])

From analyzing Karnaugh Maps [21][22], the worst case SOP for an n-bit input requires 2^{n-1} n-input minterms. If the LUTs in Figure 31 are configured to be 4-input LUTs, product minterms can be formed n bits wide, requiring $\left\lceil \frac{n}{4} \right\rceil$ LUTs per product term. Since the number of minterms required for a worst case logic function is 2^{n-1}, then the entire SOP circuit requires $2^{n-1} \cdot \left\lceil \frac{n}{4} \right\rceil$ LUTs, or $2^{n-2} \cdot \left\lceil \frac{n}{4} \right\rceil$ slices. The propagation delay is $t_{prop} = t_{net,1} + t_{LUT4} + t_{MUXCY,S \to O} + \left\lceil \frac{n}{4} \right\rceil \cdot t_{MUXCY,I0 \to O} + 2^{n-1} \cdot t_{ORCY}$, see

Appendix C.2 for explanation of terms. These equations are used by HUandDelay to estimate propagation delay and hardware utilization.

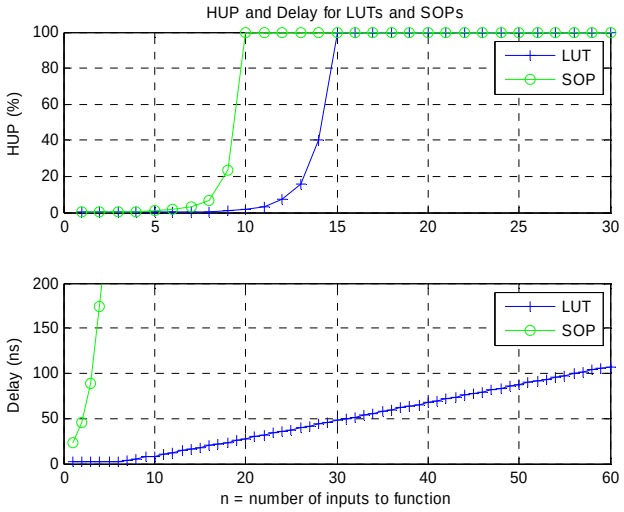


Figure 32 HUP and Propagation Delay for *n*-input LUTs and *n*-input worst case SOP.

After analyzing the estimations in Figure 32, it is apparent that when the actual function being realized is not known, it is much more appropriate to use LUTs instead of SOPs. However, when specific functions are reduced to small SOPs, the worst-case SOPs are not implemented, and a significant speed-up can occur with a reduction in hardware utilization. Consider a function that can be reduced to a sum of 4 midterms, where each minterm has 16 inputs (Figure 31). The number of slices required is $4 \times \left\lceil \frac{n}{4} \right\rceil = 16$ LUTs, or 8 slices. The corresponding HUP is 0.0079%. The propagation delay $t_{prop} = t_{net,1} + t_{LUT4} + t_{MUXCY,S \to O} + \left\lceil \frac{n}{4} \right\rceil \cdot t_{MUXCY,I0 \to O} + 4 \cdot t_{ORCY} = 2.924ns$. The same function implemented using a 16-bit LUT requires 4 BRAMs, or a HUP=0.93%, with a delay 7.06ns. Thus, when specific *n*-input functions are known and can be reduced to SOP, it may be much more efficient than using an *n*-input LUT.

6. Address Encoders/Segment Index Encoders (SIEs)

Address encoders are used in NFGs as Segment Index Encoders (SIEs) for NFGs with non-uniform segmentation. They determine in which segment an input variable ***x*** lies, and thus determines the memory location of the coefficients used in NFG calculations. The inputs to the encoder may be all or just some of the bits of the input variable ***x***. It is much more difficult to estimate hardware utilization and propagation delay for an SIE, because the size depends on two variables: the number of input bits, *n*, and the number address lines for the coefficients table, *k*. The SIE is referred to as an *n:k* SIE.

a. Architectures

The most generic address encoder is shown in Figure 33. SIEs are not required for NFGs that use constant width segmentation because appropriate bits of ***x*** can be used as address lines to the coefficient memory [12]. For NFGs with non-uniform segmentation, the number of segments required $s_{\min}$ is determined by segmentation algorithms. Segmentation algorithms take into account the function being realized by the NFG, the number of system bits, and the required accuracy of the system. They return the number of segments $s_{\min}$ and the appropriate coefficients to be stored in the NFG's coefficient table. The architecture of the Virtex-II requires memory sizes to be a power of 2 when using BRAMs. Thus a particular NFG should use $s = 2^k$ segments, where k is the number of address lines to the coefficient memory, and $k = \lceil \log_2 s_{\min} \rceil$. A detailed discussion about segmentation methods can be found in [5].

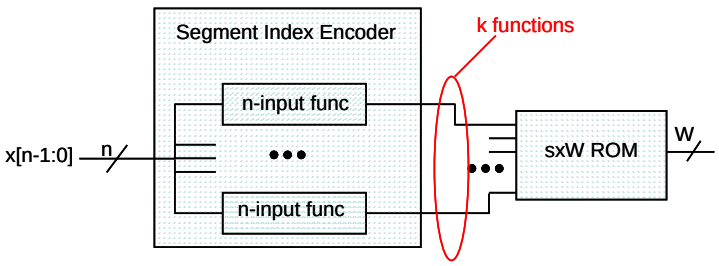


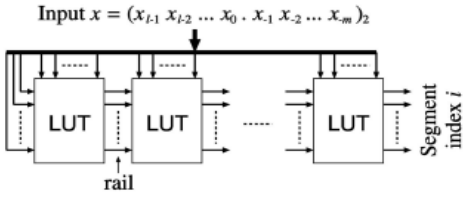
Figure 33 Generic Address Encoder.

A generic address encoder requires at most k n-input functions, for an n-bit wide $\boldsymbol{x}$. For most common NFGs, this generic method would consume an enormous amount of hardware. The size of an n-bit function is $O\left(2^n\right)$, thus a generic address encoder built in this manner would be $O\left(2^n \lceil \log_2 s \rceil\right)$. Consider an NFG with a 16-bit input $\boldsymbol{x}$ that requires s=1024 segments, or k =10. HUandDelay estimates that each 16-input function uses 2.78% of the BRAMs. This means that the SIE requires 27.8% of the BRAMs. Now consider an NFG with a 24-bit input and the same number of segments. The number of BRAMs required per function is 711.1% of the total available BRAMs. Therefore, 10 functions require 7111% of the BRAMs. In fact, an NFG with 1024 segments cannot be implemented on the Virtex-II XC2V6000 unless x is less than 18 bits long. Implementing a general address encoder using a SOP structure is impractical as well, since the worst-case number of required slices is $2^{n-2} \cdot \left\lceil \frac{n}{4} \right\rceil$. An SOP for a 16-bit input single-bit output requires 193.9% of the slices on a Virtex-II XC2V6000 FPGA.

Since it is impractical to construct a reasonably large SIE from k n-input functions or even from a SOP architecture, it is better to estimate general SIEs using LUT cascades [10][12][14][15]. LUT cascades require $2^{k+1} \times k(n-k)$ memory bits, where $k = \lceil \log_2 s_{\min} \rceil$ and $s_{\min}$ is the number of segments. The savings in hardware comes from the size being $O(n)$, instead of $O\left(2^n\right)$ for a general n-input k-output function. The general architecture of a LUT cascade is shown in Figure 34.

Interval	Index
$s_0 \le x \le e_0$	0
$s_1 < x \le e_1$	1
⋮	⋮
$s_{t-1} < x \le e_{t-1}$	$t-1$

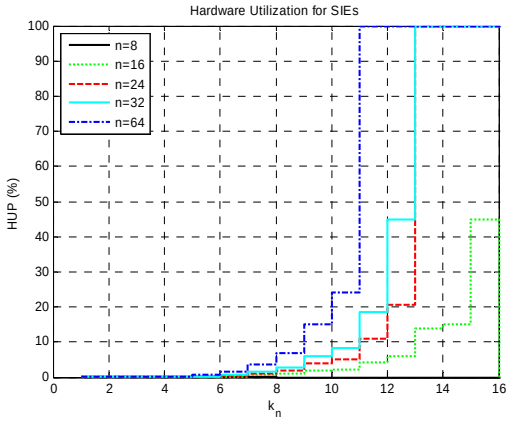
(a)



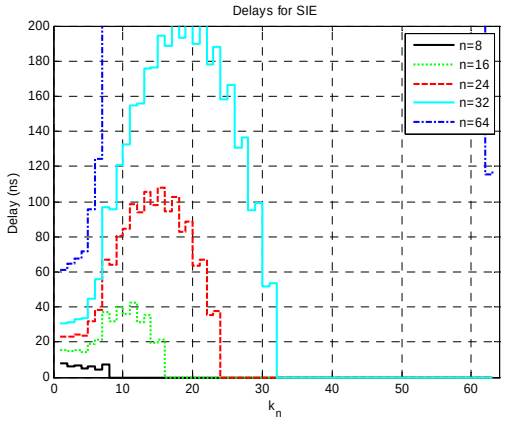
(b)

Figure 34 LUT Cascade Architecture. (From: [10][11])

The number of inputs into each LUT in the LUT cascade are k+2, the number of rails is equivalent to the number of address lines, k. This architecture requires $\left\lceil \frac{n-k}{2} \right\rceil$ (k+2)-input k-output LUTs [11]. The function HUandDelay calculates the propagation delay of the LUT cascade by cascading $\left\lceil \frac{n-k}{2} \right\rceil$ (k+2)-input LUTs. Because k LUTs are in parallel, the net delay is adjusted because the fanout of the SIE is equal to the fanout of each LUT multiplied by k.



(a) HUP



(b) Delay

Figure 35 HUP and Delay for LUT Cascades vs. k for Various n.

b. Complexity Analysis

The author's function HUandDelay returns hardware utilization parameters based on unknown functions. Therefore, the best general designs are used to determine complexity. Since LUT cascades require less hardware than SOPs and large LUTs, HUandDelay uses the architecture described above for LUT cascades to estimate the complexity of an SIE.

c. Delay Analysis

LUT cascades also exhibit shorter propagation delays for general SIE functions than from the other architectures previously discussed. Therefore, the propagation delay estimated by HUandDelay is based on that of a LUT cascade.

7. Block RAM (BRAM) and Other Memory

Memory is utilized within NFGs for storing and retrieving coefficients for the approximation technique. Using a ROM as described above is the simplest way to get an *n*-bit addressable memory, but it may not be the fastest. The Xilinx FPGA includes 18Kbit BRAM units which can accomplish the same goals with a smaller time delay. For most NFG applications, writing to memory is not required. Using the BRAMs in read-only mode can significantly reduce the delay when compared to using LUTs or distributed RAM. Other circuit designs may utilize external RAMs but since there are a wide variety of them, it is not feasible to make estimations on them all. For this reason, external RAMs are not analyzed in this thesis.

a. Architecture

BRAM is included on the Virtex-II and is one the main resources discussed throughout this thesis. It provides a relatively large block of memory with fast connections to surrounding hardware, including the MULT18x18s. The downside of using the BRAM is that there are a limited number of them (Table 1), and the circuit adjoining the block must be arranged close to the BRAM in order to minimize the routing delay. Also, if the desired amount of RAM is less than that contained in one block, then

the rest of the block is wasted. Thus, unless BRAM is used with at least 18Kbits, then hardware is wasted. Two BRAMs in parallel combined with a 2:1 MUX form a 36Kbit RAM. Thus, the number of BRAMs used is $\left\lceil \frac{2^n}{2^{14}} \right\rceil$ and the number of levels of 2:1MUXs is $\log_2 \left\lceil \frac{2^n}{2^{14}} \right\rceil = n - 14$. The overall delay is the sum of the delay from the BRAM plus the delay of the MUX network required to implement the *n*-bit address RAM.

Although each BRAM can have at most 14 address bits, they can be configured to use fewer address bits. Using fewer address bits allows the BRAM to contain more than 1-bit per memory location. Table 5 summarizes the possible BRAM configurations. This thesis compares BRAM usage for various NFG configurations using 1-bit port data width. The BUP is dependent on the number of address bits, *n* (shown in "ADDR Bus" column in Table 5), and the word width, *w* ("Port Data Width" column in Table 5). The number of memory bits stored is $s \times w = 2^n \times w$ and is constant, where *s* is the number of segments required by the NFG and *n* is the number of address lines. Thus, when *n* is increased, *w* becomes smaller.

Table 3-12: **Port Aspect Ratio**

Port Data Width	Depth	ADDR Bus	DI Bus / DO Bus	DIP Bus / DOP Bus
1	16,384	<13:0>	<0>	NA
2	8,192	<12:0>	<1:0>	NA
4	4,096	<11:0>	<3:0>	NA
9	2,048	<10:0>	<7:0>	<0>
18	1,024	<9:0>	<15:0>	<1:0>
36	512	<8:0>	<31:0>	<3:0>

Table 5 Virtex-II BRAM Configurations for Single-port RAMs. (From [18])

b. Complexity Analysis

Since there are multiple ways to configure the BRAM for various word widths, HUandDelay determines the number of bits of memory required by the

equation $2^k \times w$. The number of BRAM blocks required is $\left\lceil \frac{\text{\# of memory bits required}}{\text{\# of memory bits per BRAM}} \right\rceil = \left\lceil \frac{2^k \times w}{16384} \right\rceil$. The required BRAM blocks are multiplexed together with a $\left\lceil \frac{2^k \times w}{16384} \right\rceil$:1 MUX. HUandDelay calls itself recursively to obtain the hardware utilization parameters for the MUX. It returns the total hardware utilization parameters by summing the two. Note that there will be some wasted hardware (MUXs) if the number of BRAM blocks is not a power of 2, but the BRAMs are not wasted.

c. Delay Analysis

Analyzing the delay is somewhat more difficult for BRAMs, since they are actually synchronous circuits and every other circuit studied so far has been combinatorial. This thesis looks at combining different arithmetic devices in series to determine the overall NFG propagation time. It does not take into account setup times and hold times that a sequential circuit would. For the purposes of this thesis, the delay of a BRAM, $t_{prop,BRAM}$, is defined as $t_{prop,BRAM} = t_{NET} + t_{BCKO}$, where the net delay depends on the fanout, and t_{BCKO} is the delay from the time the clock signal transitions to the time when the output data bits are valid. In this situation, we assume the address bits to the BRAM are stable when the clock undergoes a transition. HUandDelay uses the equation above to compute the propagation delay for the BRAM as a pseudo-combinational delay. For memories that require more than one BRAM, they are combined with an appropriate-sized MUX. HUandDelay also accounts for the required MUX delay.

E. VISUALLY REPRESENTING COMPLEXITY AND PROPAGATION DELAY

Various m-files are used to plot HU-Delay Graphs, which visually represent the components. These graphs make it easy to compare components versus size and delay at the same time, and also compare different components to see which ones take up more space. The delay axis of the HU-Delay Graph represents the timeline on which the signal

propagates through a component, or through multiple components. The HUP (vertical) axis is the measure of hardware that is utilize for a particular component or components.

The author's MATLAB functions HUPBoxes.m and boxesOrigin.m both produce HU-Delay graphs. However, boxesOrigin.m keeps the bottom-left corner of each component centered at the origin, while HUPBoxes.m arranges the components based on their dependency relationships.

1. Comparing the Same Components with Different Sizes

The HU-Delay Graphs can be helpful when comparing a specific component versus size. Figure 36 compares adders at various word-widths using the function boxesOrigin.m. For example, the delay for a 64-bit adder is approximately 5.8 ns and it uses approximately 0.032% of the hardware.

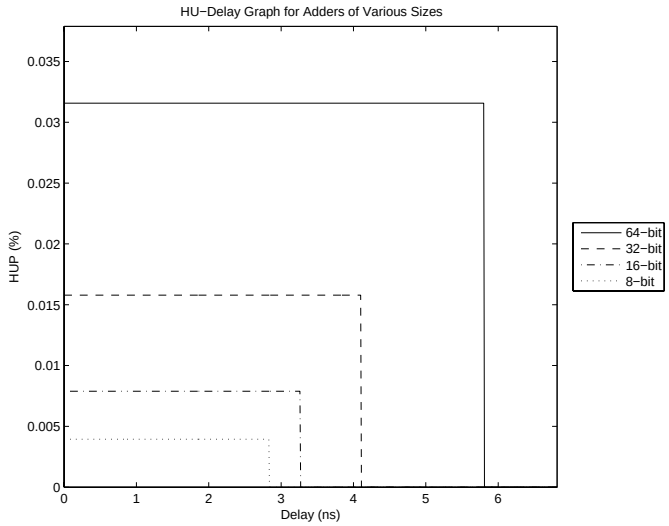


Figure 36 HU-Delay Graph of Adders with Various Word-widths.

2. Comparing Arithmetic Components with the Same Number of Input Bits

Figure 37 shows several different components with the same word width. Notice that a ROM built from CLBs with 18 address lines takes up the most space and has the longest delay, whereas the 18-bit Barrel Shifter takes the least time and least hardware. This type of comparison is useful when comparing two candidate components for a

particular NFG. For example, consider possible NFG architectures for implementing $f(x) = x^2$. One could use an 18-bit by 18-bit unsigned multiplier, while another could simply use BRAM with a total of 18 address lines. The HU-Delay graph in Figure 37 shows the comparison between the two. Notice that there are tradeoffs to consider. Using the BRAM is faster, but using the multiplier requires less hardware.

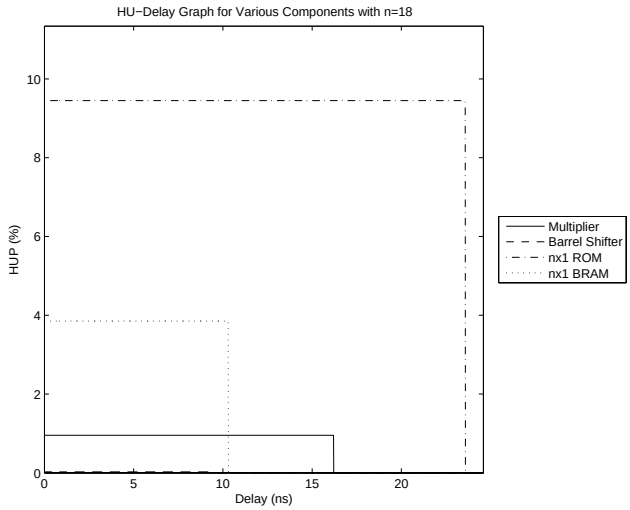


Figure 37 HU-Delay Graph of Several 18-bit Components.

3. Multiple Components in Series

Generally, NFGs contain multiple cascaded components. Linear NFGs provide a good example where the components are in series, that is, each component must wait until the previous component has completed its computation prior to initiating its own computation. Figure 38 shows an example of a linear NFG with non-uniform segmentation using the function HUPBoxes.m. The bottom-left corner of each component is anchored on the delay axis at the end of the delay of the previous component. In the example, the adder must wait until the barrel shift operation is complete; the barrel shifter must wait until the multiplier is finished; and so on. Notice that the hardware utilization for each component can be read off of the HUP axis from the top of each respective box. For example, the multiplier takes up roughly 0.95% of the FPGA hardware and the SIE takes up roughly 0.7%. The delay for each component is the width of its associated box. Thus, the SIE takes roughly 12ns to complete, while the

BRAM takes 3 to 4ns. The HU-Delay graph easily shows relative hardware utilization and delays for all of its components simultaneously.

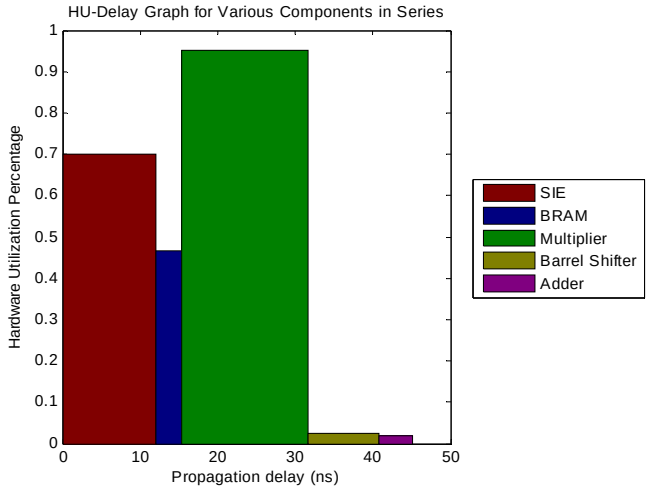


Figure 38 HU-Delay Graph of Various Components in Series.

4. Multiple Devices in Parallel

In some NFGs, calculations can be done in more than one arithmetic component at the same time. The example in Figure 39 shows the exact same components that are in Figure 38, but they are arranged in a parallel configuration. This view allows easy detection of which component takes the longest time to propagate. It also makes it easy to see the total hardware utilization for the NFG.

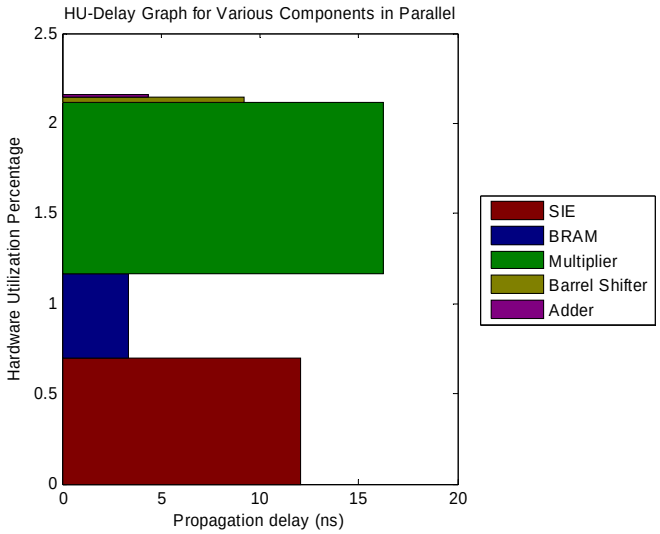


Figure 39 HU-Delay Graph of Various Components in Parallel.

5. Multiple Devices in Series/Parallel Configurations

The previous component configurations demonstrate relatively simple NFG architectures, but efficiently designed NFGs require multiple arithmetic components in a series/parallel combination. Creating HUP-Delay graphs for more complex NFGs is not as simple as the previously mentioned configurations. In order to combine multiple components, it is necessary to know what components depend on the result from other components.

The "dependency" matrix $\boldsymbol{D}$ is a square matrix that contains the dependency relationships for all of the components in a particular NFG. Each row corresponds to the particular component in the NFG. For a given NFG, let κ be the number of components in the NFG. Thus $\boldsymbol{D}$ is a $\kappa \times \kappa$ matrix. Let ρ represent the index into the list of component names, where $1 \le \rho \le \kappa$. A particular component ρ depends on another component η iff $D_{\rho,\eta} \neq 0$. Figure 40 shows an example of a simple NFG where device 2 depends on device 1, and device 3 depends on device 2. The function HUPBoxes.m uses the dependency matrix to arrange components in series and/or parallel. If a particular component is dependent on another component completing its computation, then it is said to "depend" on that component. This is particularly useful when constructing NFGs where the multipliers require an output from the memory before it can begin its computation. Thus an overall delay can be assessed if components operate in parallel. Since components can depend on more than one other component, HUPBoxes places the component in series with the component which finishes the latest, thereby computing the longest path delay. HUPBoxes disregards data in the upper right sector triangle to prevent circular dependencies.

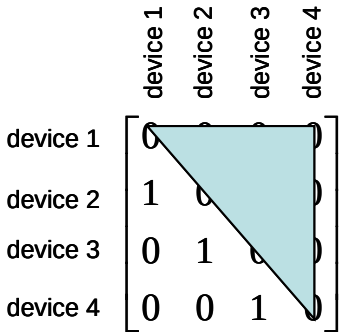
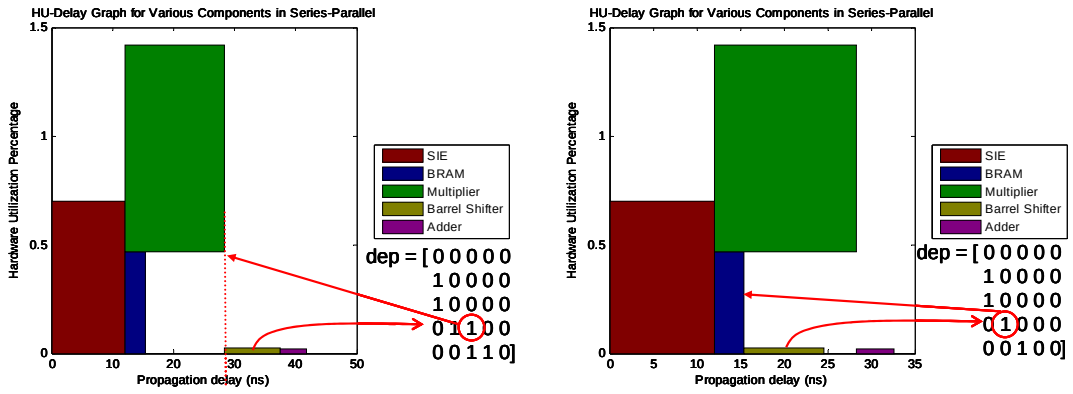


Figure 40 Example of a Dependency Matrix ***D***.

More complex NFGs are shown in Figure 41a and Figure 41b. The NFG in Figure 41a shows an NFG whose multiplier and BRAM both depend on the SIE. The barrels shifter depends on both the multiplier and the BRAM, and therefore must wait until both of them have completed their computation. Since the multiplier takes longer, then the barrel shifter starts after the multiplier is done. In the example in Figure 41b, the barrel shifter depends on the BRAM and **not** the multiplier, thus it can operate in parallel with the multiplier. Also notice that the adder must wait on the multiplier.



(a) Barrel Shifter Depends on Multiplier (b) Barrel Shifter Depends on BRAM

Figure 41 HU-Delay Graph of Series-Parallel Composite Device.

F. CHAPTER SUMMARY

This chapter shows how various arithmetic and logic components (such as multipliers and coefficient tables) can be built from the resources on the Virtex-II FPGA (CLBs, BRAMs, and MULT18x18s). It defines terminology for measuring the usage of

each resource to be used in comparing components and NFGs. This chapter also shows how simulation results are collected and how fillLin is used to estimate missing data points. This allows relatively accurate complexity and delay estimations for components that were not simulated. The hardware utilization and delay estimations for the components computed by HUandDelay are validated in this chapter. The following chapter organizes several components into specific NFG models, using the complexity and delay estimations for each component to produce complexity and delay estimations for each entire NFG. Not all of the components in this chapter are used in the models in Chapter IV. For example, MUXs and barrel shifters are not used. They were analyzed in anticipation of alternative NFG models. Future work might explore the benefits of using barrel shifters instead of multipliers in NFGs. The next chapter describes eight NFG models that are commonly described in other resources [8][11][12].

IV. CONSTRUCTING MODELS FOR CURRENT NFG ARCHITECTURES

This chapter outlines how models are constructed to accurately represent particular NFGs. The models below are simple examples of what can be constructed from the basic components listed in Table 4. The term "component" is used throughout this thesis to refer to a basic arithmetic device that is used within an NFG. For example, the components of an LUB NFG are a ROM, a multiplier, and an adder. The models use simple assumptions and estimates to reduce the number of variables determining the complexity and delay of a particular NFG.

A. NFG MODEL CONSTRUCTION AND USAGE

The models in this chapter produce HUP and delay estimations based on two known variables: the system size, *n*, and the number of required segments, *s*. The model input variable *n* determines the width of the arithmetic components and contributes in estimating the SIE if it is required. The input variable *s*, along with the size of the word stored in memory *w*, determine the size of the required memory. It also contributes to determining the size of the SIE (if required). Each model defines *w* based on the architecture and *n*.

This allows any particular model to be independent of a particular function. Generally, *s* depends on *n*, but the models do not calculate a value for *s*. Each model is only based on the particular NFG architecture and the required memory size. The architecture provides the type and quantity of arithmetic components required, the sizes of each component, and the dependency relationship between the components. For each component in the NFG, the models (i.e. model_*.m) call the author's function HUandDelay.m to retrieve the SUP, MUP, BUP, and delay. For example, if an NFG architecture requires a 17-bit adder, then the model_* file calls HUandDelay, which returns the parameters for a 17-bit adder. Each model then assembles a matrix ***C*** containing the HUPs and delays for all of its components. A corresponding dependency matrix ***D*** is also constructed within each model based on the dependency relationship

characterized by the architecture. These matrices are passed together with a list of component names into either HUPboxes.m or totalHUPandDelay.m. Both functions return the total HUP and delay along the worst case delay path. The only difference between the two functions is that the latter does not produce an HU-Delay Graph.

A feature of the MATLAB code file architecture is that any hardware configuration can be implemented as long as it uses the basic components in HUandDelay.m. Any of the models can realize any function, as long as the number of segments is known. In fact, for the same architecture, the only difference between an NFG realizing $f(x)$ and one realizing $g(x)$ is the set of coefficients stored in memory. The number of coefficients is proportional to the number of segments, which depends on the properties and domain of the function being realized by the NFG. Therefore, the size of the memory and SIE (if required) depend on the function realized on the NFG. But again, the only inputs into HUandDelay.m are *s* and *n.*

B. ESTIMATING THE APPROPRIATE SIZE FOR COMPONENTS

To make accurate size and delay estimations for NFGs, it is imperative that the estimates for its components be accurate as well. This section describes the assumptions made in order to produce a few common NFG architectures.

1. Estimating the Memory and SIE Sizes

Memory and SIE sizes are based on the number of segments, *s*, required. The number of segments depends on the function, the function interval, the type of NFG (linear, quadratic, or other) and the precision of the number system. The function segments.m calculates the number of segments required when given a function, interval, and number system size. It assumes the allowable error is $\varepsilon = 2^{-n-1}$. Higher accuracies require more segments.

The number of segments has been determined for several functions and for a few commonly-used precisions [4][8][11][13][20]. Some of the data has been collected from experimental data and some has been calculated with asymptotic approximations.

Relevant data is combined together in 0. The data can be useful, but it only provides data for three values of ε. It does not provide a general formula for various architecture sizes. 0 shows the number of segments required for linear uniform (LU), linear non-uniform (LN), quadratic uniform (QU), and quadratic non-uniform (QN) NFGs of various *n*-bit systems. Here, "uniform" and "non-uniform" refer to the segmentation type.

#	f(x)	Interval	# of Segments $\varepsilon = 2^{-17}$				# of Segments $\varepsilon = 2^{-24}$				# of Segments $\varepsilon = 2^{-33}$			
			LU	LN	QU	QN	LU	LN	QU	QN	LU	LN	QU	QN
1	2^x	$[0,1]$	89	75	8	7	Note1	849	39	35	22717	19008	311	278
2	1/x	$[1,2]$	128	75	17	10	Note1	849	81	50	32773	18996	646	400
3	$\sqrt{x}$	$[1,2]$	Note1	35	7	5	Note1	388	33	24	Note1	8729	257	189
4	$1/\sqrt{x}$	$[1,2]$	79	50	11	8	Note1	565	55	36	20066	12684	439	288
5	$\log_2(x)$	$[1,2]$	109	76	13	9	Note1	853	64	44	27833	19097	506	351
6	ln(x)	$[1,2]$	91	63	12	8	Note1	710	56	39	23171	15927	448	311
7	$\sin(\pi x)$	$\left[0,\frac{1}{2}\right]$	143	109	14	12	Note1	1227	70	58	36397	27361	559	461
8	$\cos(\pi x)$	$\left[0,\frac{1}{2}\right]$	143	109	14	12	Note1	1227	70	58	36397	27361	559	459
9	$\tan(\pi x)$	$\left[0,\frac{1}{4}\right]$	143	73	18	12	Note1	822	88	58	36397	18371	704	459
10	$\sqrt{-\ln x}$	$\left[\frac{1}{512},\frac{1}{4}\right]$	2507	207	794	33	Note1	2356	4017	163	641600	47188	34483	1312
11	$\tan^2(\pi x)+1$	$\left[0,\frac{1}{4}\right]$	285	152	30	16	Note1	1721	151	79	72793	38087	1204	631
12	$(x-1)\log_2(1-x)$ $-x\log_2 x$	$\left[\frac{1}{256},1-\frac{1}{256}\right]$	34787	314	399	37	Note1	3556	2013	183	Note1	76334	16667	1459
13	$\frac{1}{1+e^{-x}}$	$[0,1]$	28	20	5	4	Note1	226	23	20	6989	5087	178	158
14	$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$	$\left[0,\sqrt{2}\right]$	81	53	11	9	Note1	595	52	45	20696	13312	412	357
15	$\sin(e^x)$	[0,2]	Note1	449	125	54	Note1	5099	627	265	Note1	101065	5103	2121

Note 1. Data not available for these NFGs.

Table 6 Function Suite Including the Number of Segments for LN, LU, QU, and QN NFGs. (After [20][4])

The minimum segment width for a linear NFG is $\sigma_{\min}^{lin} = 4\sqrt{\frac{\varepsilon}{\left|f^{(2)}(x^*)\right|}}$ where x^* is the value at which $\left|f^{(2)}(x)\right|$ is maximum [5]. For a quadratic NFG, $\sigma_{\min}^{quad} = 4\sqrt[3]{\frac{3\varepsilon}{\left|f^{(3)}(x^*)\right|}}$. Thus for NFGs with uniform segmentation, the number of segments can be determined by dividing the domain of the NFG by the smallest segment width. Therefore $s_{\min} = \frac{b-a}{\sigma_{\min}}$, where [*a*,*b*] is the domain of the NFG. For NFGs with non-uniform segmentation, it is more complicated to determine the number of segments. The number of segments for a linear NFG with non-uniform segmentation is $s_{\min}^{LN} = s(\varepsilon) \approx \frac{1}{4\sqrt{\varepsilon}}\int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}dx$. This is derived in [4]. We also consider the number of segments for a quadratic NFG with non-uniform segmentation to be given by the analogous equation $s_{\min}^{QN} = s(\varepsilon) \approx \frac{1}{4}\frac{\int_a^b \sqrt[3]{\left|f^{(3)}(x^*)\right|}dx}{\sqrt[3]{\varepsilon}}$ (after [4]). This has not yet been proven, but is shown to be accurate by correlating it with experimental segmentation methods.

The author's m-file segments.m uses MATLAB's symbolic toolbox to calculate the derivatives above. It then substitutes values for the interval [*a*,*b*] and the number of bits, *n* to calculate the maximum segment width $\sigma_{\max}$. Since MATLAB's symbolic toolbox cannot compute the exact integrals for some of the more complicated functions (especially those using the absolute value), a numerical integration using a trapezoidal approximation [24] is implemented. The numerical integral approximation was compared to the symbolic integrations (of those able to be integrated), yielding the exact same results. The number of segments calculated with these equations matches most of the data in 0, confirming that it accurately calculates the number of required segments for a QN. Table 7 shows the results of the calculations. The values that do not exactly match those in 0 are noted below Table 7. Although there are

small differences, they are all relatively accurate. Also, since $k = \lceil \log_2 s_{\min} \rceil$, where $k \in \mathbb{Z}$, the actual number of segments being implemented is rounded up to the nearest power of two.

Numerical integration allows us to integrate any function as long as it is continuous and bounded over the given interval. Since it is used to calculate the integral of a 2nd or 3rd order derivative, we must ensure that the original function being implemented on the NFG is twice or thrice differentiable, for linear or quadratic NFGs, respectively. This makes sense, since if *f(x)* is a linear function, then it is implemented exactly with a linear NFG. Its 2nd derivative is 0; the integral of which is also 0. This yields a segment width of ∞, and 0 segments. The function segment.m allows any function input in the form of a string (for example, 'exp(x)'). The function must be recognized by the functions in MATLAB and must be a single-variable function of *x*. The domain [*a*,*b*] for the NFG is also input to yield the number of require segments, $s_{\min}$.

The function segments.m estimates the number of segments $s_{\min}$ for LU, LN, QU, and QN NFGs in a single function call. From this, each model determines the number of address lines associated with its required coefficients table, $k = \lceil \log_2 s_{\min} \rceil$. These are needed to determine the size of the memory and the size of the SIE for the NFG that realizes the specific function over a specific interval [*a*,*b*]. The HUP and delay of the most compact memory unit is returned by calling HUandDelay(*k*,'Mem',*w*), where *w* is the width of the word stored at each memory location. Each model with non-uniform segmentation also requires an *n:k* SIE.

c	Function f(x)	Interval	# of Segments $\varepsilon = 2^{-17}$				# of Segments $\varepsilon = 2^{-24}$				# of Segments $\varepsilon = 2^{-33}$			
			LU	LN	QU	QN	LU	LN	QU	QN	LU	LN	QU	QN
1	2^x	$[0,1]$	89	75	8	7	1004[3]	849	39	35	22714[1]	19196[1]	311	277[1]
2	1/x	$[1,2]$	128	75	16	10	1449[3]	849	81	50	32768[1]	19196[1]	646	400
3	$\sqrt{x}$	$[1,2]$	46[3]	35	7	5	512[3]	388	32[1]	24	11586[3]	8769[1]	256[2]	189
4	$1/\sqrt{x}$	$[1,2]$	79	50	11	8	887[3]	565	55	36	20067	12771[1]	438[1]	287[1]
5	$\log_2(x)$	$[1,2]$	109	76	13	9	1230[3]	853	64	44	27831	19291[1]	506	351
6	ln(x)	$[1,2]$	91	63	12	8	1024[3]	710	56	39	23171	16061[1]	448	311
7	$\sin(\pi x)$	$\left[0,\frac{1}{2}\right]$	143	109	14	12	1609[3]	1227	70	58	36397	27762[1]	558[1]	461
8	$\cos(\pi x)$	$\left[0,\frac{1}{2}\right]$	143	109	14	12	1609[3]	1227	70	58	36397	27762[1]	558[1]	460[1]
9	$\tan(\pi x)$	$\left[0,\frac{1}{4}\right]$	143	73	18	12	1609[3]	822	88	58	36397	18580[1]	703[1]	458[1]
10	$\sqrt{-\ln x}$	$\left[\frac{1}{512},\frac{1}{4}\right]$	4933[2]	209[1]	793[1]	33	55806[3]	2358[1]	3995[1]	163	1262744[2]	53340[1]	31957[2]	1302[1]
11	$\tan^2(\pi x)+1$	$\left[0,\frac{1}{4}\right]$	285	153[1]	30	16	3217[3]	1721	151	79	72793	38926[1]	1202[1]	629[1]
12	$(x-1)\log_2(1-x)$ $-x\log_2 x$	$\left[\frac{1}{256},1-\frac{1}{256}\right]$	1730	315[1]	398[1]	37	19564[3]	3557[1]	2006[1]	182[1]	442676[3]	80480[1]	16047[2]	1455[1]
13	$\frac{1}{1+e^{-x}}$	$[0,1]$	28	20	5	4	309[3]	226	23	20	6985	5101[1]	178	158
14	$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$	$\left[0,\sqrt{2}\right]$	81	53	11	9	915[3]	595	52	45	20697[1]	13453[1]	412	356[1]
15	$\sin(e^x)$	$[0,2]$	1221[3]	451[1]	125	53[1]	13813[3]	5101	626[1]	265	312553[3]	115402[1]	5006[1]	2116[1]

1. Slightly different from 0, but there is no difference in implemented memory sizes.
2. Different from 0, resulting in an additional address line to the implemented memory.
3. New results.

Table 7 Number of Segments Based on Proven [5] and Assumed Equations.

2. Estimating Multiplier Size

The goal of this thesis is to estimate general NFG complexity and delay without having to perform a lengthy synthesis. The multipliers analyzed in HUandDelay are *n*-bit by *n*-bit multipliers whose product is 2*n*-bits in length. Some NFG designs may

require *n*-bit by *m*-bit multipliers, where $m \neq n$. To save all data bits, the product must contain $n+m$ bits. In these cases $\left\lceil \frac{n+m}{2} \right\rceil$-bit multipliers are used because their complexities are slightly more than multipliers optimized for specific *n* and *m* value. This estimate provides a worst case estimate for a multiplier. Multiplier complexity can also be reduced by neglecting some of the output bits. For example, some NFG designs may simply require an *n*-bit multiplier with an *n*-bit product. Again, a full *n*-bit multiplier is substituted, representing a worst case multiplier size.

3. Estimating Adder Size

The adders analyzed in this thesis have two *n*-bit inputs, and produce an *n*-bit sum. However, quadratic NFGS often require multiple-input adders. Since HUandDelay does not provide information on multiple-input adders, the models in this thesis use adders in series. Also, when two inputs are different sizes, the adder uses the larger of the two sizes. Figure 42 shows an example of a 3-input adder with (m+1)-bit and (n+1)-bit inputs where $m \geq n$.

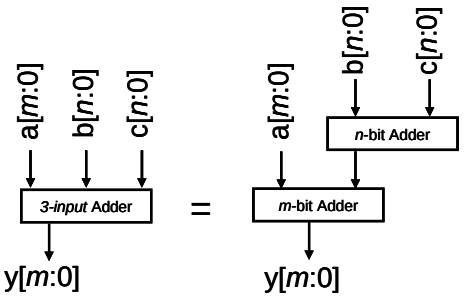


Figure 42 Using Two 2-Input Adders to Realize a 3-input Adder.

4. Estimating Other Components Not Analyzed by HUandDelay

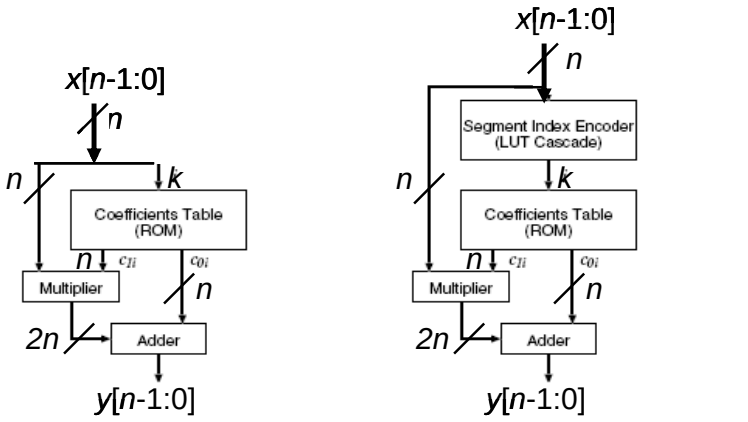
NFGs may require additional arithmetic components that are not analyzed by HUandDelay.m. For functions with few inputs (n= 1 to 7 bits) LUTs can be used to realize a general function. This may be applicable to NFGs that incorporate special number handling, or signed number manipulation. It might also be efficient to use a SOP implementation. The models in this thesis do not require special hardware.

C. MODELS FOR COMMON NFG ARCHITECTURES

The models described in this section are summarized in Table 8. They have been developed from architectures in [8][11][12]. Appendix A.1 shows how to use the models to obtain desired data and plot HU-Delay Graphs.

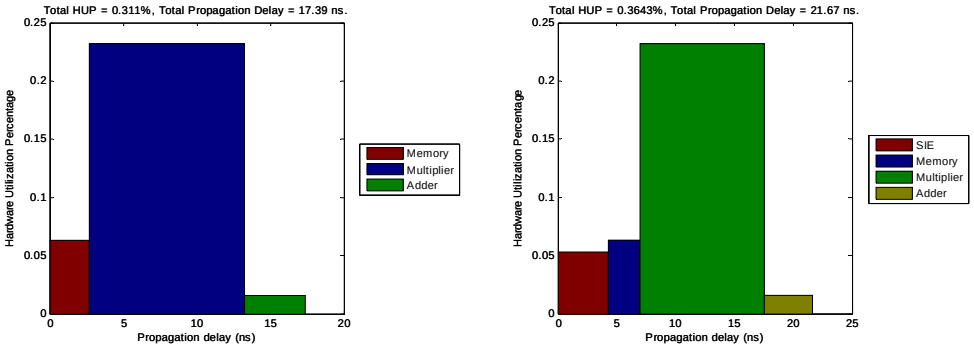
1. Basic Linear NFGs

Basic linear NFGs approximate $f(x)$ with s equations in the form $y_i(x) = c_{1i}x + c_{0i}$, where $i \in$ and $1 \le i \le s$. The constants c_{1i} and c_{0i} are stored in memory or in LUTs. The sizes of the components in the basic NFG architectures are the minimum required sizes such that no bits are truncated or rounded. For example, a multiplier with 2 *n*-bit inputs produces a product that has 2*n*-bits. The architectures are shown in Figure 43. The HU-Delay graphs in Figure 44 shown examples of basic linear NFGs realizing $f(x) = \sqrt{x}$ on [1,2].



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 43 Basic Linear NFG Architectures. (After [12])



(a) LUB (b) LNB

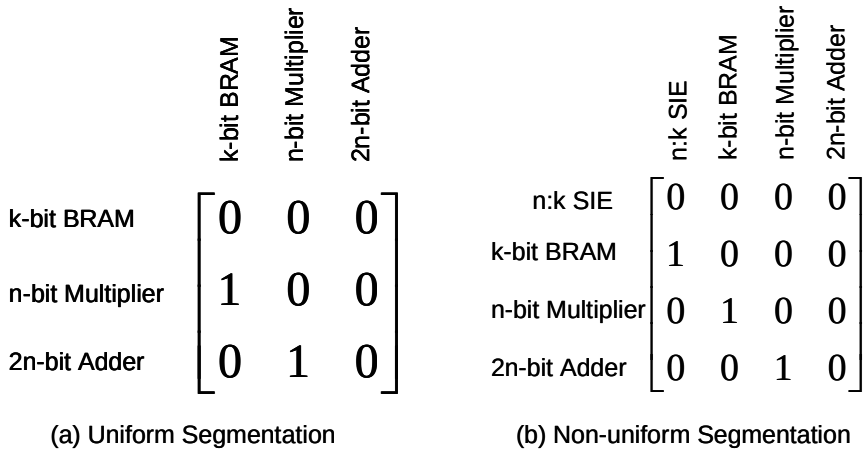
Figure 44 HU-Delay Graphs for LUB and LNB NFGs realizing $f(x) = \sqrt{x}$ on the interval [1,2] with n=16.

a. Uniform Segmentation

The architecture for a basic linear NFG with uniform segmentation (LUB) is shown in Figure 43a. It requires a $2^k \times w$ memory, an n-bit multiplier, and a $2n$-bit adder. This architecture requires two coefficients to be stored in memory for each segment. Thus, $w = 2n$. The number of segments is determined by the segments.m, and the number of address lines required for the coefficients table is $k = \lceil \log_2 s \rceil$. The multiplier requires a coefficient c_{1i} from the memory. Thus, computing c_{1i} can only occur after a memory read has been completed. Likewise, the adder must wait until the multiplier has completed its computation. Thus, the adder depends on the multiplier. This dependency is shown in the dependency matrix shown in Figure 45.

b. Non-uniform Segmentation

The basic linear NFG with non-uniform segmentation is referred to as the LNB. The only difference between architecture with non-uniform versus uniform segmentation is that the non-uniform architecture also requires an *n:k* SIE. The memory must wait for the SIE to complete its address computation before the memory can begin to look up the coefficients. The dependency is also shown in Figure 45. In general, non-uniform architectures require fewer segments. Thus, *k* is smaller than that of a similar architecture with uniform segmentation.



(a) Uniform Segmentation (b) Non-uniform Segmentation

Figure 45 Dependency Matrices for Basic Linear NFGs.

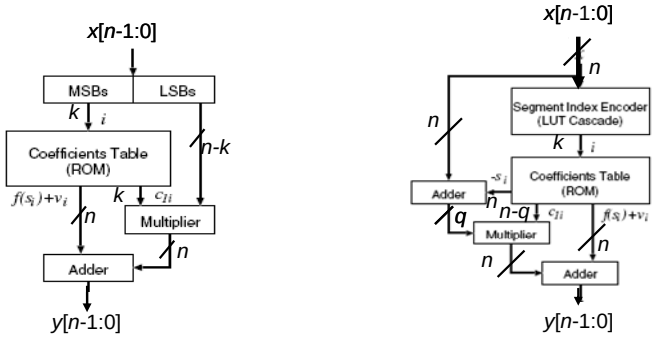
To implement a specific function with a basic linear NFG, the user must call the function model_Linear_Uniform_Basic or the function model_Linear_NonUniform_Basic with the size of the number system (*n*) and the number of segments ($s_{\min}$). The author's MATLAB m-file segments.m returns the number of segments required based on the proofs in [4] and a system error $\varepsilon = 2^{-n-1}$.

2. Compact Linear NFGs

Compact linear NFG architectures are shown in Figure 46 for both uniform and non-uniform segmentation. HU-Delay graphs are shown in Figure 47 for $f(x) = \sqrt{x}$ on [1,2]. They compute the function $y = c_{1i}\left(x - s_i\right) + f\left(s_i\right) + v_i$. These types of NFGs can be used to reduce the size of the arithmetic components. This often reduces the delay and sometimes the hardware utilization for the NFG. They do not always reduce the overall amount of hardware required. However, compare the architecture of the NFG in Figure 46a with the basic linear NFG in Figure 43a. The multiplier in the compact NFG is a *k*-bit by (*n*-*k*)-bit multiplier, resulting in an *n*-bit product. This thesis approximates this type of multiplier with a $\left\lceil \frac{n}{2} \right\rceil$-bit by $\left\lceil \frac{n}{2} \right\rceil$-bit multiplier, which is obviously smaller than the *n*-bit by *n*-bit multiplier used in the basic linear NFG above. Also the memory would

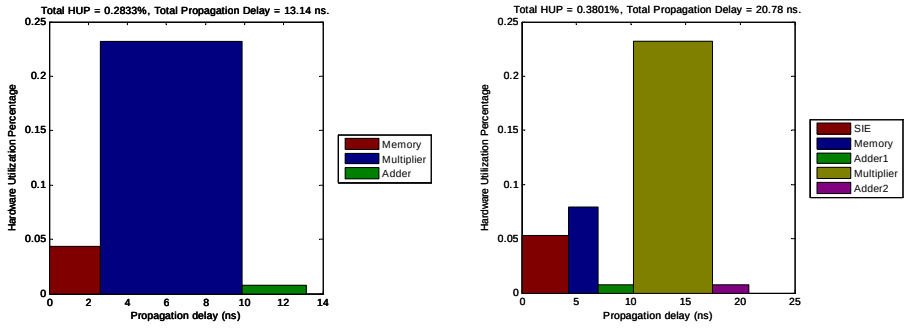
only have to store an $(n+k)$-bit word for each segment instead of a $2n$. For the architecture in Figure 46b, additional hardware is required when compared to basic linear NFG in Figure 43b: an n-bit adder and an additional coefficient in memory. Therefore there is a trade-off to be considered. The adder causes a relatively small delay and takes up very little hardware. However, if the number of segments is large, then adding an additional n-bit word for each segment can become extremely costly in terms of hardware utilization.

In addition, the architectures below must be analyzed carefully for each particular function before determining which bits may be truncated without loss of precision. Thus, q cannot be determined as a generality even though some specific architectures have been analyzed in depth [13]. To show general comparisons, the compact models in this thesis use $q = \frac{n}{2}$.



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 46 Compact Linear NFG Architectures. (After [11])



(a) LUC (b) LNC

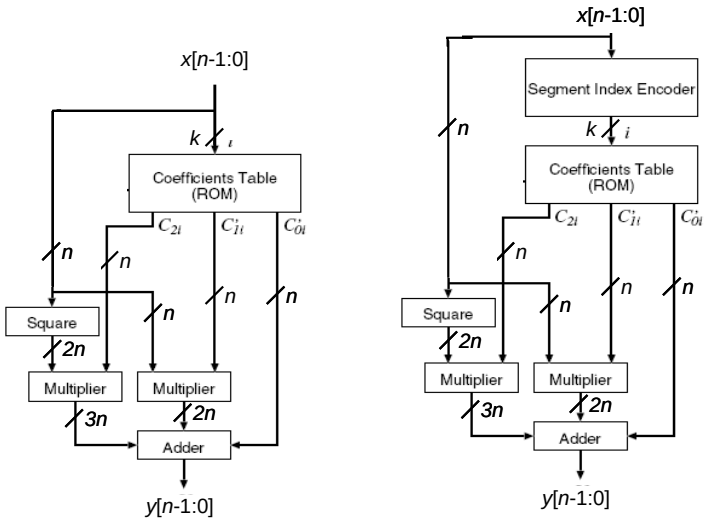
Figure 47 HU-Delay Graphs for LUC and LNC Realizing $f(x) = \sqrt{x}$ on the Interval [1,2] with n=16.

Models for the LUC and LNC return HUP and delay by calling model_Linear_Uniform_Compact and model_Linear_NonUniform_Compact respectively. The summary of the components and dependency matrices for compact linear NFGs using uniform and non-uniform segmentation methods (LUC and LNC) are shown in Table 8.

3. Basic Quadratic NFGs

Basic quadratic NFGs approximate $f(x)$ with s equations in the form $y = c_{2i}x^2 + c_{1i}x + c_{0i}$, where $i \in$ and $1 \le i \le s$. The constants c_{2i}, c_{1i}, and c_{0i} are stored in memory or in LUTs. Like the basic linear NFGs, the sizes of the components in the basic quadratic architectures are the minimum required sizes such that no bits are truncated or rounded.

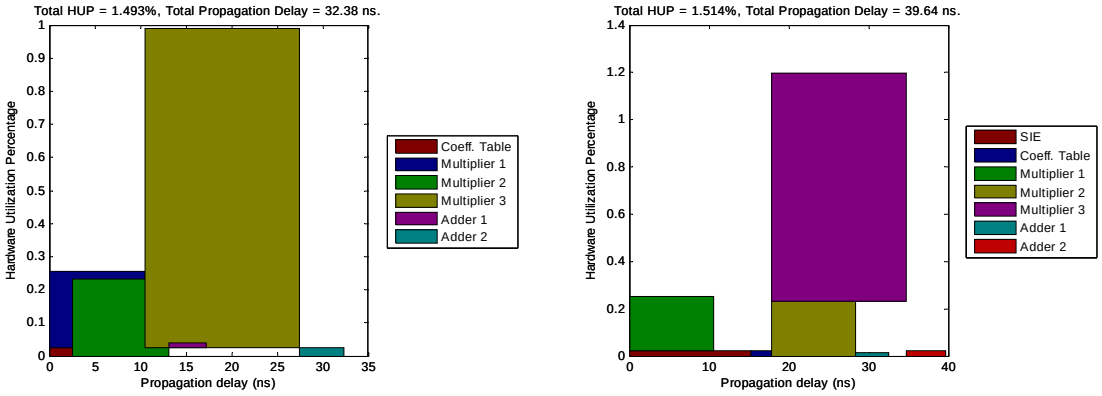
Basic quadratic architectures are shown in Figure 48 for NFGs using uniform and non-uniform segmentation. Each requires three multipliers, two adders, and a coefficients table that contains three n-bit words. The NFG with non-uniform segmentation also requires an $n{:}k$ SIE. An n-bit multiplier is used to produce x^2, which is a $2n$-bit product. To prevent truncation of any bits, a total of two n-bit multiplier and a single $1.5n$-bit multiplier are used. In addition, the first adder requires a $2n$-bit input ($c_{1i}x$) and an n-bit input (c_{0i}). Thus a $2n$ adder is used. The $2n$-bit sum ($c_{1i}x + c_{0i}$) is added to the $3n$-bit product $c_{2i}x^2$ in a $3n$-bit adder.



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 48 Basic Quadratic NFG Architectures. (After [8])

Models for the QUB and QNB return HUP and delay by calling model_Quad_Uniform_Basic or model_Quad_NonUniform_Basic. A summary of the components and dependency matrices for QUB and QNB are shown in Table 8. The HU-Delay graphs for QUB and QNB NFGs realizing $f(x)=\sqrt{x}$ on [1,2] are shown in Figure 49.

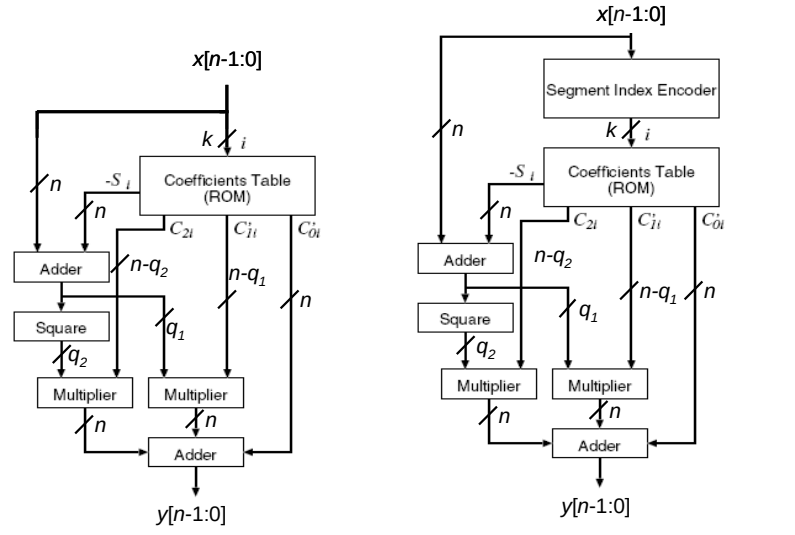


(a) QUB (b) QNB

Figure 49 HU-Delay Graphs for QUB and QNB NFGs Realizing $f(x)=\sqrt{x}$ on the Interval [1,2] with n=16.

4. Compact Quadratic NFGs

The models for compact quadratic NFGs (model_Quad_Uniform_Compact and model_Quad_NonUniform_Compact in Appendix A.2) use the basic components that are necessary to compute $y = c_{2i}\left(x - s_i\right)^2 + c_{1i}\left(x - s_i\right) + f\left(s_i\right) + v_i$, for uniform and non-uniform segmentations, respectively. Like compact linear NFGs, compact quadratic NFGs use scaling methods [7] to reduce the size of the multipliers. It is much more complex to determine the sizes of the components because they also depend on the required accuracy of the NFG. Larger multipliers can provide more precise results because fewer bits are truncated. The bit widths illustrated in Figure 50 are only an example. The sizes cannot be generalized because they depend on the system accuracies and the effects of truncating bits with respect to a particular function. Thus, they are not analyzed in the thesis, although the model can be easily modified to apply to a particular architecture with known component sizes. In depth analyses have been done in [13] for exactly rounded quadratic NFGs. The models implemented in this thesis set $q_1 = q_2 = \frac{n}{2}$ for general comparisons.



(a) Uniform Segmentation (b) Non-Uniform Segmentation

Figure 50 Compact Quadratic NFGs. (After [8])

A summary of the components and dependency matrices for QUB and QNB are shown in Table 8. The HU-Delay graphs for these two architectures are shown in Figure 51.

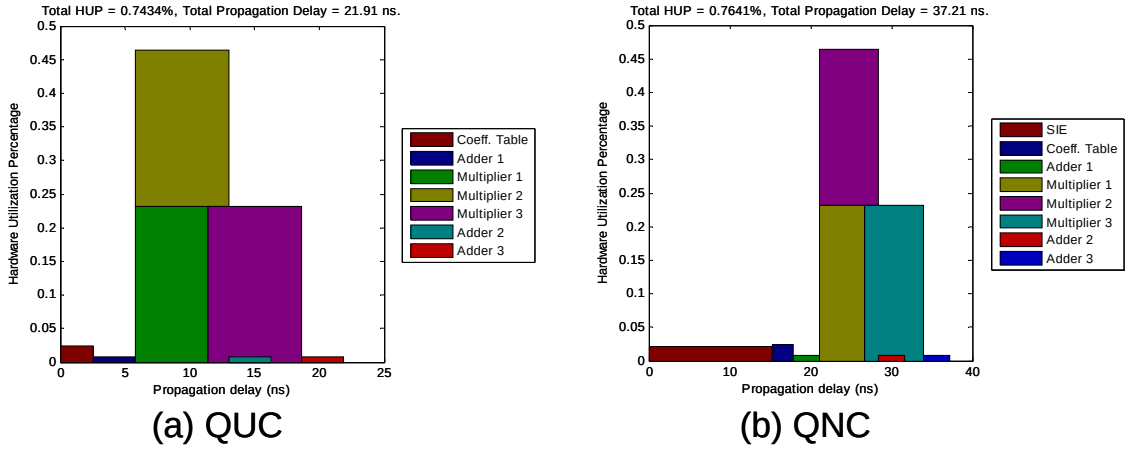


Figure 51 HU-Delay Graphs for QUC and QNC NFGs Realizing $f(x) = \sqrt{x}$ on the Interval [1,2] with n=16.

Arch #	Architecture	Component List	Dependency Matrix
1	LUB $y_i(x) = c_{1i}x + c_{0i}$ Figure 43a	$\begin{bmatrix} 2^k \times 2n \text{ Memory} \\ n\text{-bit Mult18x18} \\ 2n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$
2	LNB $y_i(x) = c_{1i}x + c_{0i}$ Figure 43b	$\begin{bmatrix} n:k \text{ SIE} \\ 2^k \times 2n \text{ Memory} \\ n\text{-bit Mult18x18} \\ 2n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
5	QUB $y = c_{2i}x^2 + c_{1i}x + c_{0i}$ Figure 48b	$\begin{bmatrix} 2^k \times 2n \text{ Memory} \\ n\text{-bit Mult18x18} \\ n\text{-bit Mult18x18} \\ 1.5n\text{-bit Mult18x18} \\ 2n\text{-bit Adder} \\ 3n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$
6	QNB $y = c_{2i}x^2 + c_{1i}x + c_{0i}$ Figure 48b	$\begin{bmatrix} n:k \text{ SIE} \\ 2^k \times 2n \text{ Memory} \\ n\text{-bit Mult18x18} \\ n\text{-bit Mult18x18} \\ 1.5n\text{-bit Mult18x18} \\ 2n\text{-bit Adder} \\ 3n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$
3	LUC $y = c_{1i}(x - s_i) + f(s_i) + v_i$ Figure 46a	$\begin{bmatrix} 2^k \times (n+k) \text{ Memory} \\ (n/2)\text{-bit Mult18x18} \\ n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$
4	LNC $y = c_{1i}(x - s_i) + f(s_i) + v_i$ Figure 46b	$\begin{bmatrix} n:k \text{ SIE} \\ 2^k \times (3n-q) \text{ Memory} \\ n\text{-bit Adder} \\ (n/2)\text{-bit Mult18x18} \\ n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$
7	QUC $y = c_{2i}(x - s_i)^2 + c_{1i}(x - s_i) + f(s_i) + v_i$ Figure 50a	$\begin{bmatrix} 2^k \times (4n - q_1 - q_2) \text{ Memory} \\ n\text{-bit Adder} \\ (q_2/2)\text{-bit Mult18x18} \\ (n/2)\text{-bit Mult18x18} \\ (n/2)\text{-bit Mult18x18} \\ n\text{-bit Adder} \\ n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$
8	QNC $y = c_{2i}(x - s_i)^2 + c_{1i}(x - s_i) + f(s_i) + v_i$ Figure 50b	$\begin{bmatrix} n:k \text{ SIE} \\ 2^k \times (4n - q_1 - q_2) \text{ Memory} \\ n\text{-bit Adder} \\ (q_2/2)\text{-bit Mult18x18} \\ (n/2)\text{-bit Mult18x18} \\ (n/2)\text{-bit Mult18x18} \\ n\text{-bit Adder} \\ n\text{-bit Adder} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$

Table 8 NFG Model Components and Dependencies.

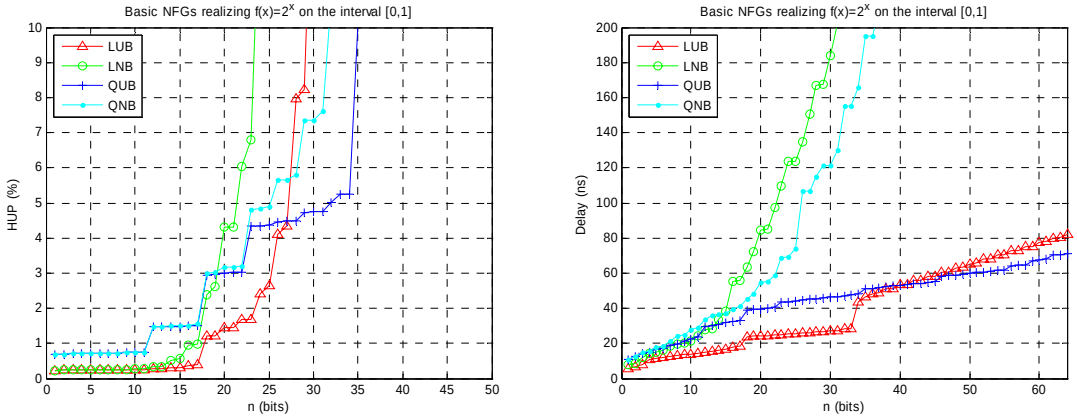
D. CHAPTER SUMMARY

This chapter shows how components are organized to form various models that represent particular NFG architectures. It shows the assumptions made for choosing the size of each component within each model. This chapter uses the complexity and delay estimations from the Chapter IV to estimate the complexity and delay for each NFG model. Future models can be constructed in similar manner with components sized specifically for particular NFGs. The models constructed in this chapter are compared in the following chapter to determine the best segmentation and approximation methods for particular functions. The next chapter analyzes complexity and delay trends for eight NFG architectures and 15 functions over a wide range of NFG sizes.

THIS PAGE INTENTIONALLY LEFT BLANK

V. COMPARING COMMON NFG ARCHITECTURES

This chapter compares the basic and compact NFGs models to determine best configuration for each model for each size. The first function in Table 7 ($f(x) = 2^x$) is used as an example in this section but Appendix D contains the same plots for all of the functions in the function suite in 0. Figure 52 shows HUP and delay versus *n* for the four basic NFG architectures realizing the function $f(x) = 2^x$ on the interval [0,1].

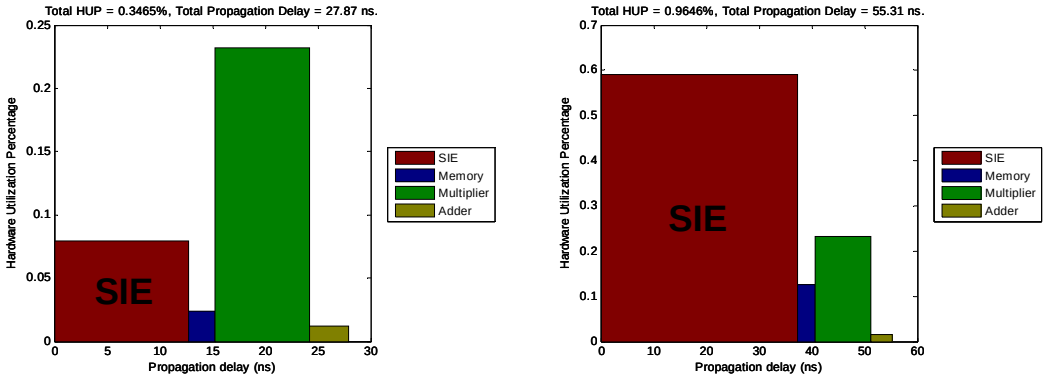


(a) HUP (b) Delay

Figure 52 Basic Architecture Comparison for NFGs Realizing $f(x) = 2^x$.

A. COMPARING UNIFORM VERSUS NON-UNIFORM SEGMENTATION

The benefits of using non-uniform segmentation can be seen in Table 7 by the reduction in the number of required segments. This results in a smaller memory size than the same NFG using uniform segmentation. However, the main reason the hardware of uniform segments is less than for non-uniform segments is the SIE. It can be seen in Figure 53 that even for small NFGs, the SIE can consume more resources and take longer than all of the other NFG components combined. As *n* gets larger, the portion of the HUP and delay that is due to the SIE grows.

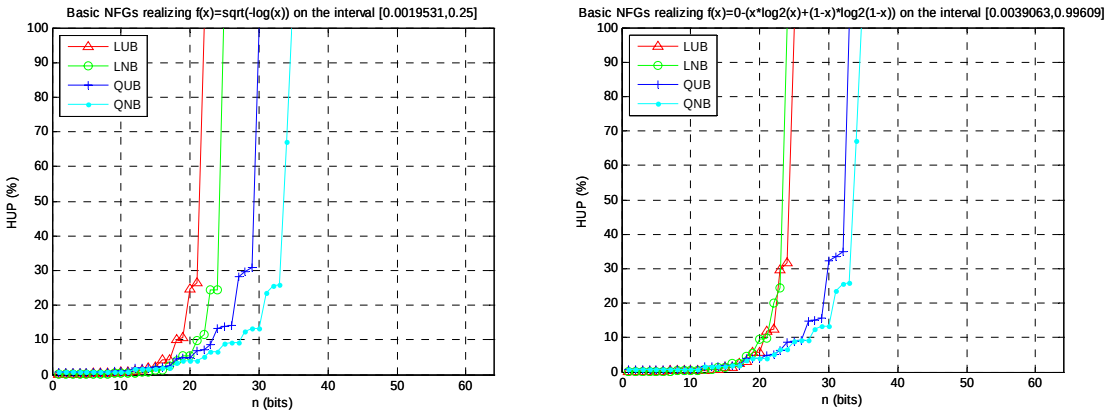


(a) n=12 bits (b) n=16 bits

Figure 53 HU-Delay Graphs for $f(x) = 2^x$ for n=12 and n=16 bits.

1. Comparing Hardware

Figure 52 clearly shows that for $f(x) = 2^x$, $HUP_{LUB} \le HUP_{LNB}$ and $HUP_{QUB} \le HUP_{QNB}$ for all n. Also $t_{LUB} < t_{LNB}$ and $t_{QUB} < t_{QNB}$ for all n. The savings in memory by using non-uniform segmentation is generally counteracted by the size and delay of the SIE that is required. Thus, in almost all cases it is better to use uniform segmentation. 13 of the 15 functions in 0 yield this result (Appendix D). The functions that do not behave the same are function 10 $\left(f(x) = \sqrt{-\ln x}\right)$ and function 12 $\left(f(x) = (x-1)\log_2(1-x) - x\log_2 x\right)$. Figure 54 shows that for function 10, non-uniform segmentation using an SIE requires less hardware than uniform segmentation for both linear and quadratic NFGs. It also shows that for function 12, non-uniform segmentation requires less hardware only in quadratic NFGs.



(a) Function 10 (b) Function 12

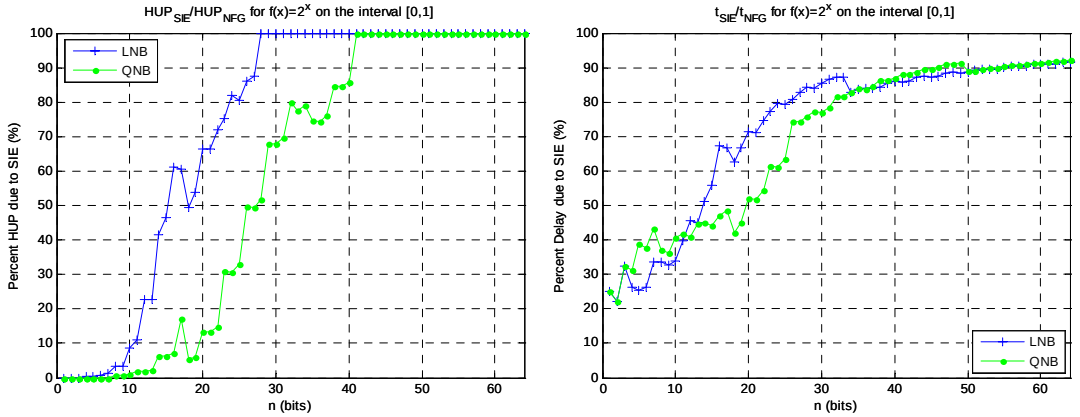
Figure 54 Cases Where Non-uniform Segmentation is Requires Less Hardware than Uniform Segmentation.

The main factor is the number of segments, which is mostly affected by the function properties. Part of Table 7 is shown in Table 9. For function 1 $\left(f(x) = 2^x \right)$, $s^{LN} \approx s^{LU} \times 84\%$ and $s^{QN} \approx s^{QU} \times 89\%$. Compare these memory savings to those for function 10, where $s^{LN} \approx s^{LU} \times 4.2\%$ and $s^{QN} \approx s^{QU} \times 4.1\%$. Here, non-uniform segmentation drastically reduces the required number of segment, *s*, so much that the combined hardware for the SIE and memory for a non-uniform NFG is less than that of the memory required for a uniform NFG. This explains why for both linear and quadratic NFGs, non-uniform segmentation requires less hardware (Figure 54a). For function 12, $s^{LN} \approx s^{LU} \times 18.2\%$ and $s^{QN} \approx s^{QU} \times 9\%$. Notice that the savings in memory is less for linear NFGs than it is for quadratic NFGs. The graph in Figure 54b shows this as well. In fact, non-uniform segmentation only benefits quadratic functions because there is a bigger reduction in the number of required segments.

#	Function f(x)	Interval	# of Segments $\varepsilon = 2^{-17}$				# of Segments $\varepsilon = 2^{-24}$				# of Segments $\varepsilon = 2^{-33}$			
			LU	LN	QU	QN	LU	LN	QU	QN	LU	LN	QU	QN
1	2^x	$[0,1]$	89	75	8	7	1004	849	39	35	22714[1]	19196[1]	311	277[1]
10	$\sqrt{-\ln x}$	$\left[\frac{1}{512},\frac{1}{4}\right]$	4933[2]	209[1]	793[1]	33	55806	2358[1]	3995[1]	163	1262744[2]	53340[1]	31957[2]	1302[1]
12	$(x-1)\log_2(1-x)$ $-x\log_2$	$\left[\frac{1}{256},1-\frac{1}{256}\right]$	1730	315[1]	398[1]	37	19564[1]	3557[1]	2006[1]	182[1]	442676	80480[1]	16047[2]	1455[1]

Table 9 Functions with a Large Number of Segments.

Figure 55 shows how much of the NFG hardware is consumed by the SIE alone for NFGs with non-uniform segmentation for $f(x) = 2^x$. Note that SIEs generally contribute to at least 20% of the total NFG delay for a small *n*, and over 90% of the delay for larger *n*. For a 16-bit LNB NFG, over 50% of the NFG hardware complexity is in the SIE. The majority of a 28^+-bit QNB NFG is also made up of the SIE alone. Graphs for the other functions in Table 7 display similar characteristics for NFGs with non-uniform segmentation. These are shown in Appendix D.



(a) Hardware consumed by SIE (b) Delay due to SIE

Figure 55 Percent Hardware Utilization and Delay due to SIE for $f(x) = 2^x$.

We now seek a criterion to determine when it is better to use uniform segmentation and when it is better to use non-uniform segmentation. Specifically, we seek to establish the crossover point between these two based on hardware utilization. In order to understand where the crossover occurs, we must examine the NFG components

closely. The components for an *n*-bit LUB NFG are exactly the same as a LNB except for the memory size and the LNB requires an *n:k* SIE. Here we will analyze the differences between the two architectures.

For a given function $f(x)$, let $s^{non-unif} = 2^{\lceil \log_2 s_{\min}^{non-unif} \rceil}$ and $s^{unif} = 2^{\lceil \log_2 s_{\min}^{unif} \rceil}$ be the number of segments required for non-uniform and uniform segmentation, respectively. They depend on the particular function and interval as well as the required precision. The values for $s_{\min}^{non-unif}$ and $s_{\min}^{unif}$ are known for the various precisions of the 15 functions in Table 7. They can also be computed by using the author's function segments. Define the Segment Reduction Ratio (SRR) to be $SRR = \frac{s_{\min}^{non-unif}}{s_{\min}^{unif}}$. The *SRR* represents the number of segments required for an NFG with non-uniform segmentation compared to uniform. The number memory bits required for the LUB NFG is $M_{unif} = 2^{k_u} \times w$, where $k_u = \lceil \log_2 s_{\min}^{unif} \rceil$ and the word size stored at each memory location is $w = 2n$ for a LUB (or $w = 3n$ for a QUB). Let $M_{non-unif}$ be the memory bits required to realize the SIE and the coefficients table for the non-uniform NFG. Thus, $M_{non-unif} = \left\lceil \frac{n-k_n}{2} \right\rceil 2^{k_n+2} \cdot k_n + 2^{k_n} \times w$, where $k_n = \lceil \log_2 s_{\min}^{non-unif} \rceil$. This assumes that the coefficients table contains a power of 2 memory locations. A non-uniform NFG requires more hardware than a uniform NFG, when $M_{non-unif} \geq M_{unif}$. Now define SSR_{crit} to be the value of *SRR* when $M_{non-unif} = M_{unif}$, or

$$\left\lceil \frac{n-k_n}{2} \right\rceil 2^{k_n+2} \cdot k_n + 2^{k_n} \times w = 2^{k_u} \times w .$$

Let $SRR = SRR_{crit}$ and substitute $s_{\min}^{unif} = \frac{s_{\min}^{non-unif}}{SRR_{crit}}$ into $k_u = \lceil \log_2 s_{\min}^{unif} \rceil$, therefore

$$k_u = \left\lceil \log_2 \frac{s_{\min}^{non-unif}}{SRR_{crit}} \right\rceil = \left\lceil \log_2 s_{\min}^{non-unif} + \log_2 \frac{1}{SRR_{crit}} \right\rceil .$$

Since we assume that $s^{non-unif} = 2^{\lceil \log_2 s_{\min}^{non-unif} \rceil}$, ($s^{non-unif}$ is an integer power of 2), then

$$k_u = \left\lceil \log_2 s^{non-unif} + \log_2 \frac{1}{SRR_{crit}} \right\rceil = \log_2 s^{non-unif} + \left\lceil \log_2 \frac{1}{SRR_{crit}} \right\rceil.$$

Therefore,

$$\left\lceil \frac{n-k_n}{2} \right\rceil 2^{k_n+2} \cdot k_n + 2^{k_n} \times w = 2^{\log_2 s^{non-unif} + \left\lceil \log_2 \frac{1}{SRR_{crit}} \right\rceil} \times w = 2^{k_n} 2^{\left\lceil \log_2 \frac{1}{SRR_{crit}} \right\rceil} \times w$$

Dividing both sides of the equation by 2^{k_n} yields,

$$4\left\lceil \frac{n-k_n}{2} \right\rceil \cdot k_n + w = 2^{\left\lceil \log_2 \frac{1}{SRR_{crit}} \right\rceil} \times w = \left\lceil \frac{1}{SRR_{crit}} \right\rceil \times w$$

Knowing that $\left\lceil \frac{1}{SRR_{crit}} \right\rceil \geq \frac{1}{SRR_{crit}}$,

$$4\left\lceil \frac{n-k_n}{2} \right\rceil \cdot k_n + w \geq \frac{1}{SRR_{crit}} \times w$$

Solving for SRR_{crit} yields

$$SRR_{crit} \geq \frac{w}{4\left\lceil \frac{n-k_n}{2} \right\rceil \cdot k_n + w}$$

This equation is plotted in Figure 56 for basic linear and basic quadratic NFGs. Now we seek to find the minimum value of SRR_{crit}. First consider the case where *n* is even. Since $k_n, n \in \mathbb{Z}$ and k_n is even, $n-k_n$ is even. Thus, we can remove the ceiling function.

$$SRR_{crit}\big|_{n=even} \geq \frac{w}{(2n-2k_n)k_n + w} = \frac{1}{(2n-2k_n)\frac{k_n}{w} + 1}$$

Therefore,

$$SRR_{crit}\Big|_{n=even} \geq \frac{1}{2n\frac{k_n}{w} - 2\frac{k_n^2}{w} + 1}$$

For basic linear NFGs, $w = 2n$. For basic quadratic NFGs, $w = 3n$. Thus,

$$SRR_{crit}^{\text{Basic Linear}}\Big|_{n=even} \geq \frac{1}{k_n - \frac{k_n^2}{n} + 1} \text{ and } SRR_{crit}^{\text{Basic Quadratic}}\Big|_{n=even} \geq \frac{1}{\frac{2k_n}{3} - \frac{2k_n^2}{3n} + 1} = \frac{3/2}{k_n - \frac{k_n^2}{n} + \frac{3}{2}}$$

For cases when n is odd, $n - k_n$ is odd. Thus,

$$\left\lceil \frac{n-k_n}{2} \right\rceil = \left\lceil \frac{n-k_n-1}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil = \frac{n-k_n-1}{2} + 1$$

Therefore,

$$SRR_{crit}\Big|_{n=odd} \geq \frac{w}{4\left(\frac{n-k_n-1}{2}+1\right)\cdot k_n + w} = \frac{w}{\left(2n-2k_n+2\right)\cdot k_n + w}$$

This reduces to :

$$SRR_{crit}\Big|_{n=odd} \geq \frac{1}{2n\frac{k_n}{w} - 2\frac{k_n^2}{w} - 2\frac{k_n}{w} + 1}$$

For linear and quadratic cases, $w = 2n$ and $w = 3n$. Thus,

$$SRR_{crit}^{\text{Basic Linear}}\Big|_{n=odd} \geq \frac{1}{k_n - \frac{k_n^2}{n} - \frac{k_n}{n} + 1}$$

and

$$SRR_{crit}^{\text{Basic Quadratic}}\Big|_{n=odd} \geq \frac{1}{\frac{2k_n}{3} - \frac{2k_n^2}{3n} - \frac{2k_n}{3n} + 1} = \frac{3/2}{k_n - \frac{k_n^2}{n} - \frac{k_n}{n} + \frac{3}{2}}$$

Since $\frac{k_n}{n} > 0$,

$$SRR_{crit}^{\text{Basic Linear}}\Big|_{n=odd} > SRR_{crit}^{\text{Basic Linear}}\Big|_{n=even}, \text{ and } SRR_{crit}^{\text{Basic Quadratic}}\Big|_{n=odd} > SRR_{crit}^{\text{Basic Quadratic}}\Big|_{n=even}$$

Thus, the minimum critical *SRR*, $SRR_{crit,\min}$ can be found by finding the minimum of $SRR_{crit}\Big|_{n=even}$, or the maximum of $\dfrac{1}{SRR_{crit}\Big|_{n=even}}$. Differentiating the latter is much simpler and provides the same information. Thus,

$$\frac{\partial}{\partial k_n}\frac{1}{SRR_{crit}^{\text{Basic Linear}}} = \frac{\partial}{\partial k_n}\left(k_n - \frac{k_n^2}{n} + 1\right) = 0$$

Solving for k_n yields

$$1 - \frac{2k_n}{n} = 0 \Rightarrow k_n = \frac{n}{2}$$

This means that the maximum of $\dfrac{1}{SRR_{crit}\Big|_{n=even}}$ occurs when $k_n = \dfrac{n}{2}$, therefore the minimum of $SRR_{crit}\Big|_{n=even}$ occurs when $k_n = \dfrac{n}{2}$. Applying the same process to the quadratic case yields the same results. Substituting $k_n = \dfrac{n}{2}$ to find $SRR_{crit,\min}$ yields

$$SRR_{crit,\min}^{\text{Basic Linear}} \geq \frac{1}{\frac{n}{2} - \left(\frac{n}{2}\right)^2 \frac{1}{n} + 1} = \frac{1}{\frac{n}{4} + 1} = \frac{4}{n+4}$$

and

$$SRR_{crit,\min}^{\text{Basic Quadratic}} \geq \frac{3/2}{\frac{n}{2} - \left(\frac{n}{2}\right)^2 \frac{1}{n} + \frac{3}{2}} = \frac{3/2}{\frac{n}{4} + \frac{3}{2}} = \frac{6}{n+6}$$

$SRR_{crit,\min}$ is the minimum *SRR* below which non-uniform segmentation requires less hardware than uniform segmentation, regardless of k_n or k_u. Thus, $SRR_{crit,\min}$ is also independent of the number of segments, $s^{non-unif}$ and s^{unif}. It is shown that $SRR_{crit,\min}$ is

only a function of n when $s^{non-unif} = 2^{\lceil \log_2 s_{\min}^{non-unif} \rceil}$ and $s^{unif} = 2^{\lceil \log_2 s_{\min}^{unif} \rceil}$. Recall the definition, $SSR = \frac{s_{\min}^{non-unif}}{s_{\min}^{unif}}$. Also recall that for linear NFGs,

$$s_{\min}^{unif} = \frac{b-a}{\sigma_{\max}} = \frac{b-a}{4\sqrt{\frac{\varepsilon}{\left|f^{(2)}(x^*)\right|}}} \text{ and } s_{\min}^{non-unif} = s(\varepsilon) \approx \frac{1}{4\sqrt{\varepsilon}} \int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}\,dx .$$

Therefore,

$$SRR^{\text{Linear}} \approx \frac{\frac{1}{4\sqrt{\varepsilon}} \int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}\,dx}{b-a} 4\sqrt{\frac{\varepsilon}{\left|f^{(2)}(x^*)\right|}} = \frac{\int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}\,dx}{(b-a)\sqrt{\left|f^{(2)}(x^*)\right|}}$$

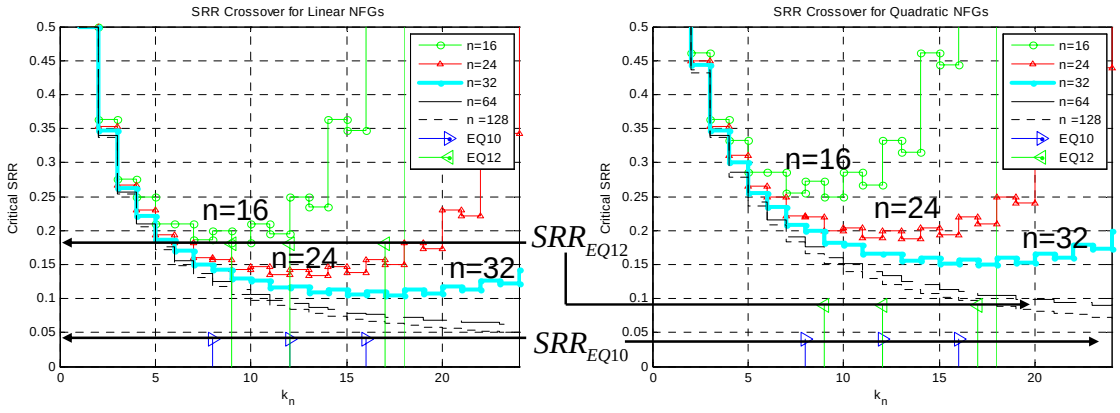
for small ε. For the analyses in this thesis, ε is sufficiently small. For all of the functions in Table 7, the maximum difference in *SRR* is 0.022. The largest ε in Table 7 is 2^{-17}. Since practical NFGs generally require $\varepsilon \le 2^{17}$, calculating the SRR for a function using the asymptotic equations above relatively accurate.

Clearly, the *SRR* of a particular NFG depends only on the function being realized and its domain [*a*,*b*], and not on ε. Therefore, *SRR* does not depend on *n*. This is also confirmed by comparing the *SRRs* in Table 10 , which are calculated from the numbers of segments in Table 7. The significance of this conclusion is that if the number of segments for a particular function is known for both uniform and non-uniform segmentations, then SRR_{crit} can be found as a function of *n* and $s_{\min}^{non-unif}$. Since the *SRR* of a particular function does not depend on *n*, the relation between SRR_{crit} and *SRR* determines at what values of *n* non-uniform segmentation is beneficial.

Once *n*, $f(x)$, and [*a*,*b*] are known, it can be determined easily if a non-uniform segmentation is always beneficial independent of the number of segments required. If $SRR^{\text{Basic Linear}} = \frac{\int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}\,dx}{(b-a)\sqrt{\left|f^{(2)}(x^*)\right|}} < \frac{4}{n+4} = SSR_{crit,\min}^{\text{Basic Linear}}$, then a linear NFG using non-uniform segmentation requires less hardware than the same NFG using uniform

segmentation. These calculations are based on using SIEs comprised of LUT cascades and using Chebyshev polynomials to compute the coefficients for each segment [5].

The results are shown in Figure 56. *SRR*s for equations 10 and 12 have also been plotted in Figure 56. There are three points for each function corresponding to the calculated values for each precision in Table 10. Notice that for equation 10 ($f(x) = \sqrt{-\ln x}$), $SRR_{EQ10} \approx 0.04$ for both linear and quadratic NFGs. These are below any of the SRR_{crit} curves shown for both the linear and quadratic NFGs for $n \leq 64$. Correspondingly, the HUP plots in Figure 54 shows that $HUP_{LNB} < HUP_{LUB}$ and $HUP_{QNB} < HUP_{QUB}$. For equation 12, $SRR_{EQ12} \approx 0.18$ for linear NFGs, lying above the SRR_{crit} curve for n > 24. This means that uniform segmentation consumes less total hardware for a 24-bit NFG realizing $f(x) = (x-1)\log_2(1-x) - x\log_2 x$ than non-uniform segmentation. This is also shown in Figure 54a



(a) Linear (b) Quadratic

Figure 56 Critical SRR for Various *n*.

#	f(x)	Interval	$SRR\left(\varepsilon = 2^{-17}\right)$		$SRR\left(\varepsilon = 2^{-24}\right)$		$SRR\left(\varepsilon = 2^{-33}\right)$	
			Linear	Quadratic	Linear	Quadratic	Linear	Quadratic
1	2^x	$[0,1]$	0.842	.875	.845	0.897	0.845	0.891
2	1/x	$[1,2]$	0.6	0.625	0.586	0.617	0.586	0.619
3	$\sqrt{x}$	$[1,2]$	0.760	0.714	.758	0.75	0.757	0.738
4	$1/\sqrt{x}$	$[1,2]$	0.63	0.727	0.637	0.655	0.636	0.655
5	$\log_2(x)$	$[1,2]$	0.697	0.692	0.693	0.688	0.693	0.694
6	ln(x)	$[1,2]$	0.692	0.667	0.693	0.696	0.693	0.694
7	$\sin(\pi x)$	$\left[0,\frac{1}{2}\right]$	0.762	0.857	0.693	0.829	0.763	0.824
8	$\cos(\pi x)$	$\left[0,\frac{1}{2}\right]$	0.762	0.857	0.763	0.829	0.763	0.824
9	$\tan(\pi x)$	$\left[0,\frac{1}{4}\right]$	.510	0.667	0.511	0.659	0.510	0.651
10	$\sqrt{-\ln x}$	$\left[\frac{1}{512},\frac{1}{4}\right]$	0.042	0.042	0.042	0.041	0.042	0.041
11	$\tan^2(\pi x)+1$	$\left[0,\frac{1}{4}\right]$	.537	0.533	0.535	0.523	0.535	0.523
12	$(x-1)\log_2(1-x)$ $-x\log_2 x$	$\left[\frac{1}{256},1-\frac{1}{256}\right]$	.182	0.093	0.182	0.091	.182	0.091
13	$\frac{1}{1+e^{-x}}$	$[0,1]$	0.714	0.800	0.731	0.870	0.730	0.888
14	$\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$	$\left[0,\sqrt{2}\right]$	0.654	0.818	0.650	0.865	0.650	0.864
15	$\sin(e^x)$	[0,2]	0.369	0.424	0.369	0.417	0.369	0.423

Table 10 Table of *SRR* for the Suite of Functions.

For the majority of the functions in Table 10, $SRR > 0.5$. This is well above all of the curves in Figure 56. This means that non-uniform segmentation results in higher hardware utilization for 13 of the 15 functions.

In summary, it is only beneficial to implement non-uniform vice uniform segmentation when it can be shown that there is a large savings in the number of required segments (small *SRR*). The minimal amount of savings SRR_{crit} is related to the number

of segments and the size of the NFG being implemented, *n*. If the coefficient tables contain a power of 2 memory locations (which is often the case in hardware), this minimum amount of savings can be quantified. The actual amount of savings $SRR_{f(x)}$ is shown to depend only on $f(x)$ and the domain of the NFG realizing it [*a,b*]. Data plots in Appendix D.1 show which particular NFG realizations require less hardware for particular functions.

The derivations of $SRR_{crit,\min}$ have been shown above for the basic architectures described in Chapter IV, but they can also be applied to other architectures. We can generalize the process by allowing *w* to remain in the equations for SRR_{crit}.

Since $SRR_{crit}\big|_{n=odd} \geq \dfrac{1}{2n\frac{k_n}{w} - 2\frac{k_n^2}{w} - 2\frac{k_n}{w} + 1} > SRR_{crit}\big|_{n=even} \geq \dfrac{1}{2n\frac{k_n}{w} - 2\frac{k_n^2}{w} + 1}$,

$SRR_{crit,\min} = \min\left(SRR_{crit}\big|_{n=even}\right)$.

Now we find the minimum of general equation:

$$SRR_{crit}\big|_{n=even} \geq \frac{1}{2n\frac{k_n}{w} - 2\frac{k_n^2}{w} + 1} = \frac{w/2n}{k_n - \frac{k_n^2}{n} + w/2n}$$

Like the linear and quadratic cases, the minimum occurs when $k_n = \dfrac{n}{2}$. Thus,

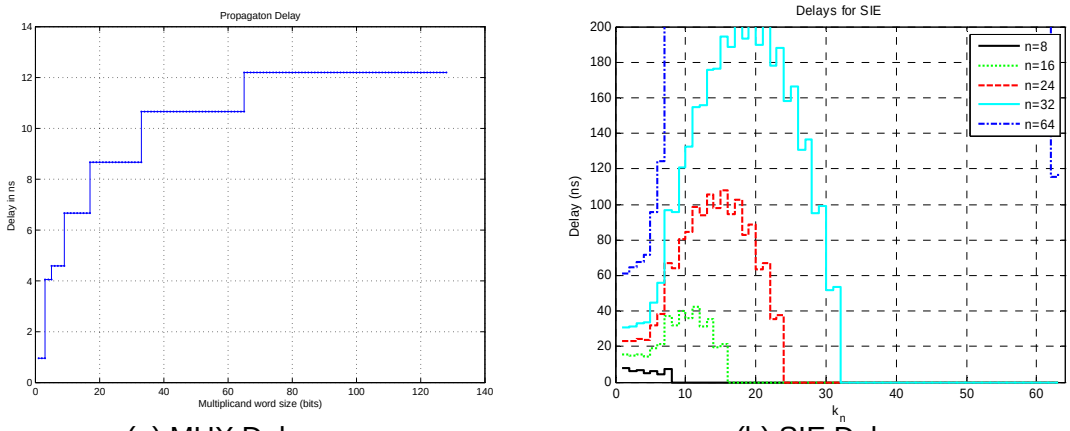
$$SRR_{crit,\min} \geq \frac{w/2n}{k_n - \frac{k_n^2}{n} + w/2n} = \frac{w/2n}{\frac{n}{4} + w/2n} = \frac{2w/n}{n + 2w/n}$$

This determination stems from a comparison between M_{unif} and $M_{non-unif}$, and assumes that the remaining arithmetic components in the two NFGs are exactly the same. For example, consider the compact NFG architectures described in Chapter IV. The compact linear NFG assumes $w = \lceil 1.5n \rceil$ and the compact quadratic NFG assumes $w = 3n$. To compare other architectures, simply replace *w* with the number of bits stored at each location in memory.

2. Comparing Delays

The delay graphs for basic and compact NFGs in Appendix D.1 show that for all of the functions in the function suite the delay is larger for NFGs with non-uniform segmentation. Figure 55b shows that at least 20% of the delay of a non-uniform NFG is due to the SIE alone. The percent delay that is attributed to the SIE is shown for 15 functions in Appendix D.4.

Again, the main difference between uniform and non-uniform NFGs is the SIE in the latter. The remaining hardware is the same, and contributes the same delay to the total delay. This section compares the delay for a coefficients table for an NFG with uniform segmentation, $t^{unif} = t_{ROM}^{unif}$, to the sum of the delays of for the coefficient table and SIE for an NFG with non-uniform segmentation, $t^{non-unif} = t_{ROM}^{non-unif} + t_{SIE}$. For $s \le 2^{14}$, or $k \le 14$, a single BRAM can be used as the coefficients table. Thus, $t_{ROM} = t_{BRAM}$. Therefore, if both $k_n \le 14$ and $k_u \le 14$, then $t_{ROM}^{unif} = t_{ROM}^{non-unif} = t_{BRAM}$ and $t^{non-unif} > t^{unif}$ for all n because of the SIE. When $k_n > 14$ and $k_u > 14$, $t^{unif} = t_{BRAM} + t_{(k_u-14):1MUX}$ and $t^{non-unif} = t_{BRAM} + t_{(k_n-14):1MUX} + t_{k_n+2:k_nSIE}$. If $k > 21$, then all of the BRAM on the Xilinx Virtex-II would be consumed. Thus the maximum required MUX size is a 7:1 MUX. Figure 57 shows that a 7:1 MUX has a delay of $t_{MUX,\max} = t_{7:1MUX} \approx 4.6ns$. To find the minimum t_{SIE} when $k_n > 14$, we look at the delay for a 16:15 SIE because n must be greater than k_n. Therefore, $t_{SIE,\min}^{k_n>14} \ge 21.6ns$. Since when $k_n > 14$ and $k_u > 14$, $t_{SIE,\min}^{k_n>14} > t_{MUX,\max}$, it follows that $t^{non-unif} > t^{unif}$ for all n.



(a) MUX Delay (b) SIE Delay

Figure 57 MUX and SIE Delays.

Since $k_u \geq k_n$ for all non-uniform NFGs, there is only one remaining case to consider: when $k_n \leq 14$ and $k_u > 14$. Here $t^{unif} = t_{BRAM} + t_{(k_u-14):1MUX}$ and $t^{non-unif} = t_{BRAM} + t_{n:k_nSIE}$. $t^{unif} > t^{non-unif}$ iff $t_{(k_u-14):1MUX} > t_{n:k_nSIE}$. Again, the maximum delay for the MUX is $t_{7:1MUX} \approx 4.6ns$. Figure 58 shows when $t_{n:k_nSIE} \leq 4.6ns$, the x-axis is k_n and the y-axis is $n - k_n$.

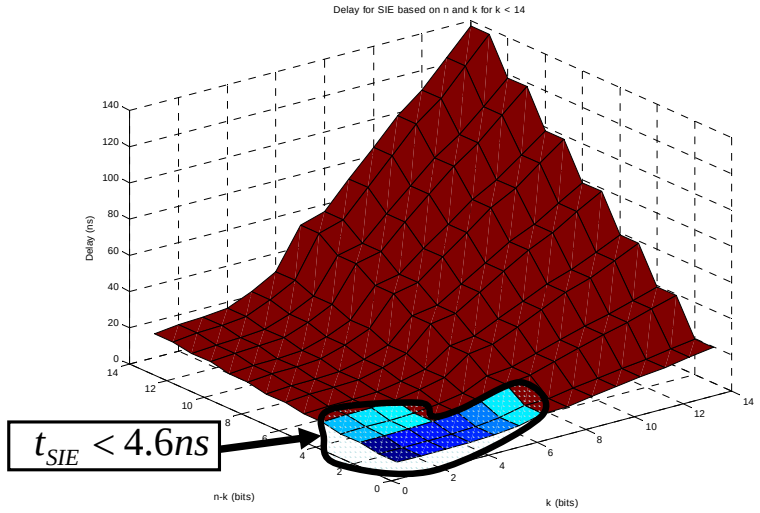


Figure 58 Delay for SIE <4.6ns.

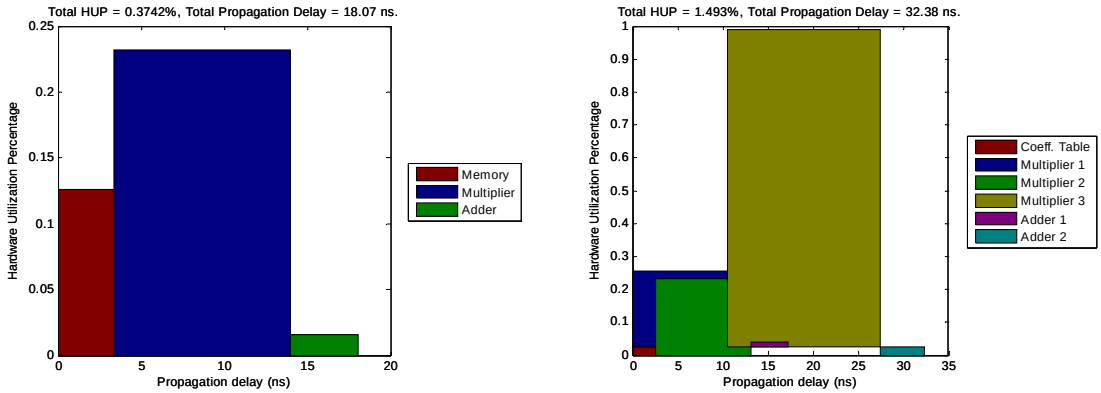
The maximum size SIE where the delay is less than that of the maximum MUX is an 8:6 SIE. This means *n* can be at most 8-bits. Therefore, when $n > 8$, an NFG with

uniform segmentation is always faster than one with non-uniform segmentation. In addition, in order for a non-uniform NFG to be faster than a uniform NFG, it would require that $k_n \le 6$ and $k_u = 21$. This means that $s^{non-unif} \le 2^6 = 64$ segments and $s^{non-unif} \approx 2^{21} = 2,097,152$ segments. Correspondingly, $SRR \le 2^{-15} \approx 0.00003$. In summary, there are not likely any practical cases where $t^{unif} > t^{non-unif}$. The plots in Appendix D.1 confirm this.

B. COMPARING LINEAR VERSUS QUADRATIC

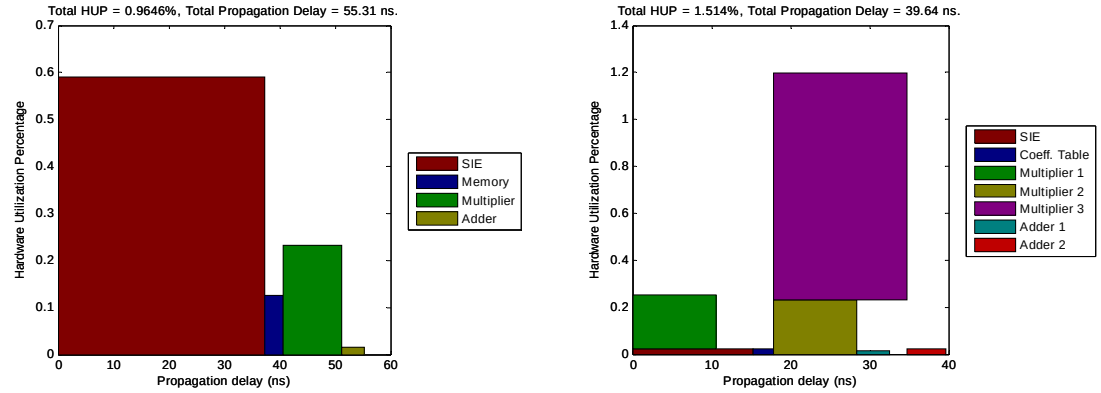
When considering whether to use quadratic or linear NFGs, there are tradeoffs to consider. The tradeoff comes between arithmetic component hardware size and coefficient table size. The size of the coefficients table depends on the function and interval. For a given function, the number of segments is less for a quadratic NFG than for a linear NFG. But the basic quadratic NFG requires three coefficients for each segment while the basic linear requires only two. Thus, the coefficient table is 150% that of the linear NFG. In addition, quadratic NFGs require additional multipliers and adders which grow in complexity as *n* grows. The tradeoff occurs when *n* gets big such that the coefficients table becomes a larger percentage of the overall NFG complexity than the rest of the arithmetic components. An example of when the crossover occurs is shown in Figure 52 for both HUP and delay. For the function $f(x) = 2^x$ on the interval [0,1], when $n < 40$, $t_{LUB} < t_{QUB}$, and when $n < 27$, $HUP_{LUB} < HUP_{QUB}$. This is only one example, but the graphs in Appendix D.1 show where the crossovers occur for the remaining 14 functions in the function suite.

The HU-Delay graphs in Figure 59 and Figure 60 compare 16-bit NFGs realizing $f(x) = 2^x$ on [0,1]. The total HUP and delay are less for the LUB than for the QUB. Clearly, the linear NFG is better. Now compare the non-uniform NFGs in Figure 60. Since the SIE makes the linear NFG much bigger and have a larger delay, the delay of the LNB is longer than that of the QNB. However, the QNB requires more hardware than the LNB.



(a) LUB (b) QUB

Figure 59 HU-Delay Graph Comparing LUB and QUB.



(a) LNB (b) QNB

Figure 60 HU-Delay Graph Comparing LNB and QNB.

In general, for large *n*, it is better to implement quadratic NFGs for a given type of segmentation. When the reduction in coefficient table size from quadratic to linear NFGs accounts for the reduction in arithmetic component complexity from linear to quadratic NFGs, then quadratic NFGs become less complex than their linear counterparts. Since memory and SIE sizes depend on the particular function, generalizing a criterion for deciding whether a linear or quadratic NFG requires more hardware, or has a longer delay, is extremely difficult. For this reason, we apply the data collected from estimations using the models in Chapter IV. The crossover points for delay and hardware utilization can be found in the graphs in Appendix D.1. The crossover points for delay and HUP often occur at separate values of *n*. This means that if it is desired to minimize hardware usage instead of the delay, then the HUP crossover must be considered.

C. CHAPTER SUMMARY

This chapter shows how the estimation tools developed in Chapter IV are used to analyze characteristics of common NFG architectures. It analyzes eight NFG models for 15 functions, providing graphical data that shows which architecture consumes the least hardware or has the smallest delay for each function. This data shows that quadratic NFGs require less hardware and have shorter delays as the size of the NFG gets larger. It also establishes a criterion for when non-uniform segmentation is beneficial for a particular function, based on the size of the NFG. The findings in this chapter show that NFGs with non-uniform segmentation generally require more hardware and almost always have longer delays than NFGs with uniform segmentation. Chapter VI summarizes the findings in this chapter and the development of the models in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

This thesis develops a software model for estimating complexity and delay for NFGs. It also uses the software to analyze characteristics of common NFGs.

A. SOFTWARE MODEL

This thesis shows how complexities and delays for NFGs can be estimated without having to build them. The software framework developed in this thesis provides a fast method for comparing NFGs over a wide range of functions, architectures, and sizes.

1. Comparing Common NFG Component Complexity and Delay

The software can be used to find hardware utilization and delay for several components. The implementations of common NFG components in specific FPGA hardware are analyzed in depth to estimate their complexity and delay based on the number of inputs, n (up to n=128). Specific simulation data from behavioral models and schematic circuits is used in determining the complexity and delay of each component. Missing data is interpolated with linear approximations. The software provides a quick and simple way to determine hardware utilization and delay for a particular component. This allows various components to be compared to determine which best suits a particular application.

2. Modeling and Comparing NFGs

This software provides a simple means to combine several components in series/parallel configurations to represent an NFG or other arithmetic logic device. The software determines the worst case propagation delay through the NFG as well as the total hardware used by the NFG. It can be used to compare various NFG architectures for various sizes. The HUP-Delay graphs can be used to visually compare NFGs, as well as visually compare the relative sizes and delays of the components inside them.

B. RESULTS OF NFG ANALYSES

The results provide an easy way to choose the best architecture based on hardware complexity and/or delay. This thesis also shows that the complexity and delay of an NFG greatly depend on the complexity and delay of its coefficient table and associated SIE (for non-uniform NFGS).

1. Benefits of Non-uniform Segmentation

For 13 of the 15 functions analyzed in this thesis, non-uniform segmentation offers no benefits. However, when non-uniform segmentation drastically reduces the number of segments in an NFG, it can reduce the overall hardware utilization. The delay is almost always longer for NFGs with non-uniform segmentation.

a. A Criterion when Non-Uniform Segmentation Requires Less Hardware

The majority of the functions in Table 10 show that non-uniform segmentation still requires at least 50% of the segments requires by uniform segmentation. Two of the fifteen functions show reductions by lower than 10%. This thesis shows a criterion that can be used to determine which segmentation method requires less hadware for basic NFGs. It compares the reduction in the number of segments by non-uniform segmentation (*SRR*) to the NFG size, *n*. The minimum amount of reduction required, $SRR_{crit,\min}$, depends on the number of segments (which depends on $\varepsilon(n)$) and the properties and domain of the function being realized. This thesis also shows that the *SRR* of a given function depends only on the properties of that function and the domain of the NFG implanting the function. When the number of segments (corresponding to the number of memory locations) is restricted to a power of two, $SRR_{crit,\min}$ becomes a function of *n* only. For a basic linear NFG, if $\frac{\int_a^b \sqrt{\left|f^{(2)}(x^*)\right|}dx}{(b-a)\sqrt{\left|f^{(2)}(x^*)\right|}} < \frac{4}{n+4}$ (or $SRR^{\text{Basic Linear}} < SSR^{\text{Basic Linear}}_{crit,\min}$), then non-uniform segmentation requires less hardware. This is true for basic quadratic NFGs when

$\frac{\int_a^b \sqrt{\left|f^{(3)}(x^*)\right|}dx}{(b-a)\sqrt{\left|f^{(3)}(x^*)\right|}} < \frac{6}{n+6}$. From these equations, a critical value of *n* can be determined, n_{crit}, below which it is always more hardware efficient to use non-uniform segmentation. The derivations of these equations assume that LUT cascades are used in the SIE for the non-uniform NFGs and Chebyshev polynomials are used to determine the coefficients for the approximation equations. They also assume the basic architectures described in Chapter IV are used.

b. Delays for Non-Uniform Segmentation

This thesis shows that non-uniform segmentation always has a longer delay than uniform segmentation, except in rare trivial NFGs (where $n \le 8$). In fact, when NFG architectures for 15 functions were compared in terms of delay, non-uniform NFGs proved the best only in a few cases when $n \le 2$. If $n \le 2$, two LUTs can be used instead of an NFG. Therefore, for all practical NFGs, propagation delay is longer when non-uniform segmentation is implemented. Appendices D.2.2 and D.3.2 show the best architectures based on delay.

2. Linear vs. Quadratic NFGs

When considering linear versus quadratic NFGs for the 15 functions in the suite, LUB NFGs consume less hardware than QUB NFGs for *n* less than $\approx$25 to 29 bits. They also have smaller delays than QUB NFGs for $n \approx 37$ to 39 bits. Appendix D.2 shows which of the four basic architectures is best in terms of HUP for all 15 of the functions in Table 7. It also shows which is better in terms of delay. The crossover points for compact architectures vary from the basic architectures. Appendix D.3 shows the best of the compact architectures.

C. RECOMMENDATIONS FOR FUTURE WORK

The method of estimating component complexity and delay in this thesis allows meaningful comparisons to be made. The software developed in this thesis is meant to be used in future applications with minor alterations.

1. Using Other FPGAs

It may be beneficial to estimate hardware utilization and delay for the models developed in this thesis on other FPGAs. The author's MATLAB file LoadXilinxDeviceData contains specifications for the Xilinx Virtex-II XC2V6000 FPGA with a speed grade of -4. The timing and hardware parameters can be specified for other Virtex-II FPGAs as well. HUandDelay assumes that arithmetic components are constructed as described in Chapter III. The method of component construction is common to all Virtex-II FPGAs. Thus, by changing the parameters in LoadXilinxDeviceData, complexity and delay estimations can be made easily for the entire family of FPGAs. To estimate FPGAs other than Virtex-II, minor alterations to HUandDelay are required to allow for variations in component construction. For example, the Virtex-II resources include 18-bit signed multipliers. Other FPGAs may not contain multipliers at all. Therefore, the multiplier estimation section has to be re-written to provide estimations based on how the specific FPGA implements multipliers.

2. Creating and Comparing Other Models

Each of the eight models in this thesis has been constructed in a standard manner. They can be used as templates to build other models.

a. Analyzing Other Methods for Reducing NFG Hardware and Delay

Modern research concerning NFGs often focuses on reducing hardware and/or delay. Research in [5] shows a reduction in the number of segments by implementing non-uniform segmentation, resulting in dramatic reduction in the amount of memory required for the NFG. Other research shows that a reduction in arithmetic

component size can be achieved by other means [4]. For example, using linear NFGs that have a slope that is a power of two reduces complex mulipliers into simpler barrel shifters. Models can easily be built to compare the tradeoffs between the several methods.

b. Comparing NFGs with Specifically Sized Components

Architectures in [13] are shown to reduce arithmetic component complexity and actually specify component bit-widths. Models for these architectures can be constructed and compared to the basic models in this thesis to illustrate relative hardware and delay savings. The size of each component in the NFG can be specified in the model file (i.e. model_*.m), allowing the models to be extremely flexible.

3. Categorizing Functions that Benefit from Non-Uniform Segmentation

This thesis shows that non-uniform segmentation is only beneficial when $SRR_{f(x)}$ is small. The values of $SRR_{f(x)}$ depend only on the function and the domain of the NFG realizing it. For linear NFGs, it is related to $f^{(2)}(x)$, and for quadratic NFGs, it is related to $f^{(3)}(x)$. Specific functions can be found where $SRR_{f(x)}$ is small. Thus, they are likely candidates to employ non-uniform segmentation.

4. Analyzing Domain/Range Reduction Methods for Reducing NFG Hardware and Delay

Aside from looking at the properties of particular functions, examining their domains may assist in reducing the number of segments, which reduces the complexity and delay of the NFG. Domain reduction methods allow the NFG's domain to be shifted where it requires fewer segements. However, they often include additional arithmetic components. Models can be constructed to conduct tradeoff analyses for these domain reduction methods so that optimal domains can be determined.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB SOURCE CODE

A.1 M-FILE USAGE

In order to use the MATLAB source code, all of the m-files in this appendix are required to be in the same folder, along with the text files that are imported (Appendix B.2). When entering commands, or calling functions, the user must be in the current directory where the m-files are stored.

1. Comparing Individual Components

To compare individual components, type the following into MATLAB's command window:

```
[SUP BUP MUP t] = HUandDelay(n,component,w)
```

This will produce the SUP, MUP, BUP and delay for the given component. The variable '*component*' is a string that matches one of the following strings: 'Adder', 'Mult','Mult18x18','MUX','RAM','ROM','BRAM','BS','SIE','Mem','CLB', or 'SOP.' The values of *n* and *w* are the input word width and output word width respectively. In some cases, the complexity and delay do not require both inputs. A summary of all of the components that can be analyzed with HUandDelay is shown in Table 4.

This function can be used to produce the hardware utilization and delays of various sized components for comparisons. To calculate the hardware utilization in a single term, the HUP of a given component can be calculated with the following command once SUP, MUP, and BUP are calculated:

```
HUP_comp = HUP(SUP, MUP, BUP)
```

2. Comparing NFG models

The HUP and delay can be found for several NFG architectures that have been implemented in models. The following commands can be used to compare various models:

```
[HUP_comp t_comp] = pickModel(ModelNum,n,s)
```

This will return the HUP and delay for an NFG with system size *n*, that requires *s* segments. The variable '*ModelNum*' can be any integer. Table 11 summarizes the models that are implemented base on the value of '*ModelNum.*'

ModelNum	NFG Model
1	LUB
2	LNB
3	QUB
4	QNB
5	LUC
6	LNC
7	QUC
8	QNC

Table 11 Model Number Index.

3. Comparing Functions

The HUP and delay can be found for any function over any interval. The number of segments must be known, or the function must meet the requirements for segment estimation discussed in Chapter IV. The functions and corresponding domains in Table 7 may be easily returned by calling the function funcSel with its input variable equal to the index number of the function. The following code shows how to get the HUP and delay for a given function on a given interval with a given system size *n.*

```
modelNum=1   % corresponds to LUB NFG
n=32              % corresponds to the system size
funcNum = 1  % corresponds to f(x)=2^x on [0,1]

[f a b] = funcSel(1)
numSegs=segments(f,a,b,n)
[HUP_NFG t_NFG] = pickModel(modelNum,n,numSegs(1))
```

This will produce the HUP and delay for LUB NFG the realizes $f(x) = 2^x$ on [*a,b*]. The variable *'funcNum'* chooses the function from the function list, and returns the functions as a string expression and the domain of the NFG [*a,b*]. If *'funcNum'* is not an integer between 1 and 15, then funcSel prompts the user to input a function and domain. Any function of x may be entered, if it is recognized as a single-variable function in MATLAB. The author's function segments returns the number of segments required in a vector corresponding to the segmentation techniques, [LU LN QU QN]. To implement a particular model for an NFG, choose the corresponding number of segments (*numSegs(1), numSegs(2), numSegs(3),*or *numSegs(4)*).

4. Producing HU-Delay Graphs

To produce a HU-Delay Graph to represent an NFG or other arrangement of components, the user must know the HUP and delay for the components. The user must also construct a dependency matrix, based on the arrangement of the components, and a list component names. Once these are determined, they are input into HUPBoxes with the following command:

```
[totHUP totDelay] = HUPBoxes(components,dependency,compNames);
```

The variable *'components'* is a matrix with two columns and a row for every component in the NFG. The first column holds the HUP value for the component corresponding to the row number. The second column holds the delay value for that particular component. The variable *'dependency'* is the dependency matrix discussed in Chapter IV. The variable *'compNames'* is an array of strings, where each row holds the

string name for the particular component. Each string (row) must be the same length in the matrix '*compNames.*' The function HUandDelay will return the total delay along the worst case path through the NFG and the overall HUP.

A.2 MATLAB FILES

1. M-file List

The following MATLAB source code was written by the author. Table 12 is the list of m-files and their dependencies.

M-file/function	Depends on
BlackLineStyle	none
boxesOrigin	BlackLineStyle
fillLin	none
funcSel	none
HUP	none
HUPBoxes	none
HUandDelay	LoadXilinxDeviceData HUP fillLin HUandDelay (Recursion) IMPORTS data from: MultDelayWithNet.txt MultSlices.txt MuxDelayWithNet.txt
LoadXilinxDeviceData	fillLin IMPORTS data from: NetDelay.txt
model_Linear_NonUniform_Basic model_Linear_NonUniform_Compact model_Linear_Uniform_Basic model_Linear_Uniform_Compact model_Quad_NonUniform_Basic model_Quad_NonUniform_Compact model_Quad_Uniform_Basic model_Quad_Uniform_Compact	HUandDelay HUP HUPBoxes totalHUPandDelay
myInt	none
pickModel	model_Linear_NonUniform_Basic model_Linear_NonUniform_Compact model_Linear_Uniform_Basic model_Linear_Uniform_Compact model_Quad_NonUniform_Basic model_Quad_NonUniform_Compact model_Quad_Uniform_Basic model_Quad_Uniform_Compact
segments	myInt symbolic\syms.m symbolic\syms.m
totalHUPandDelay	none

Table 12 M-file List with Dependencies.

2. M-file Source Codes

FILE: BlackLineStyle.m

```
function [styleCode] = BlackLineStyle(index);

%  This function returns a string variable to be used as a line style
%   Written by Tim Knudstrup, August 30, 2007

index=round(abs(index));    % ensures positive integer
numStyles = 9;
index = mod(index,numStyles);

switch index
    case 1
        styleCode='k-';
    case 2
        styleCode='k--';
    case 3
        styleCode='k-.';
    case 4
        styleCode='k:';
    case 5
        styleCode='k.:';
    case 6
        styleCode='k.-';
    case 7
        styleCode='k+-.';
    case 8
        styleCode='k*:';
    case 9
        styleCode='k*-';
    otherwise
        styleCode='k-';
end
```

FILE: boxesOrigin.m

```
function [a] = boxesOrigin(s,t)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% boxesOrigin.m                                                                  %
%                                                                                %
% This function/program plots HU-Delay Graph for various components              %
% and each component is centered at the origin.                                  %
%                                                                                %
%   function [a] = boxesOrigin(s,t)                                              %
%                                                                                %
%   Input:        s:   Vector containing Size values                             %
%                 t:   vector containing Time Delay Values                       %
%                                                                                %
%  Output:       a:    Returns 1 if no error                                     %
```

```
%                                                                               %
%  Comments:          s and t must be the same length                           %
%                                                                               %
% Created by:  Tim Knudstrup                                                    %
%       Date:  20 September 2007                                                %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%s=[2 3 4 5 6 8];
%t=[10 2 3 4 5 12];

inc=0.01;

tAxisLength=max(t)+1;
sAxisLength=max(s)+1;

tAxis=[0:inc:tAxisLength];
NumComps=max(size(t));
t_len=max(size(tAxis));
sizeMatrix=zeros(NumComps,t_len);

for comp=1:NumComps
    tcum(comp)=tAxisLength-sum(t(comp+1:end));
end

close all;
figure(1)
comp =1;
for comp=1:NumComps
    for k=1:(t_len)
        tVal=k*inc;
        if tVal <= t(comp)
            sizeMatrix(comp,k)=s(comp);
        end
    end
end

for p=1:NumComps
    p
    colr = ([rand(1) rand(1) rand(1)]).^1.5;
    plot(tAxis,sizeMatrix(p,:),BlackLineStyle(p))
    hold on
end
hold off
axis([0 tAxisLength 0 max(s)*1.2]);
legend

ylabel('HUP (%)');
xlabel('Delay (ns)');
print -depsc -tiff BoxesOrigin.eps

a=1;
```

FILE: fillLin.m

```
function [filledX filledY] = fillLin(dataX,dataY)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% fillLin.m                                                                      %
%                                                                                %
% This function creates filledX and filledY vectors containing data at           %
%   every integer ranging from 1 to the maximum integer value of of dataX        %
%   The values in filledY match those in the original dataY, and for data        %
%   points not included in dataX, filledY values are estimated using             %
%   linear approximation between the data points that do exist.                  %
%                                                                                %
%   function [filledX filledY] = fillLin(dataX,dataY)                            %
%                                                                                %
%   Input:        dataX:   X values for data points                              %
%                 dataY:   Y values for data points                              %
%                                                                                %
%  Output:      filledX:   X values from 1 to max dataX                         %
%               filledY:   Y values corresponding to filledX                     %
%                                                                                %
%  Comments:   1.   dataX must be positive integers only                         %
%              2.   dataX must be the same length as dataY                       %
%                                                                                %
% Created by:  Tim Knudstrup                                                     %
%       Date:  20 September 2007                                                 %
%                                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



% Trial DATA
%dataX = [1 2 5 9 20];
%dataY = [3 4 6 8 10];

dataX = round(dataX); % makes sure all x values are integers

unit=1;
filledX =  [1:unit:max(dataX)];
len=length(filledX);
lenData=length(dataX);
dummy=123456789;
filledY = dummy*(0*filledX+1);

filledY(1)=dataY(1);



for k=1:lenData
    filledY(dataX(k))=dataY(k);
end

k=1;
beginIndex=1;
endIndex=1;
while (k < len)&&(beginIndex<len)&&(endIndex<len)
```

```
    while (filledY(beginIndex) ~= dummy)&&(beginIndex<len)
        beginIndex = beginIndex+1;
    end
    endIndex=beginIndex;
    while (filledY(endIndex) == dummy)  &&(endIndex<len)
        endIndex = endIndex + 1;
    end
    if filledY(beginIndex)==dummy
        if beginIndex > 1
            m=(filledY(endIndex)-filledY(beginIndex-1))/(filledX(endIndex)-
filledX(beginIndex-1));
            b=filledY(beginIndex-1)-filledX(beginIndex-1)*m;
        end


        for kk=beginIndex:endIndex
            filledY(kk)=filledX(kk)*m+b;
        end
    end
    k=k+1;
    beginIndex=endIndex+1;
end

filledX=filledX';
filledY=filledY(1:len)';
%plot(dataX,dataY,filledX,filledY)
```

FILE: funcSel.m

```
function [ f a b ] = funcSel(funcNum);

%   This function returns the string representing the function
%   and its domain for one of the functions in the function suite.
%   The input variable 'funcNum' is the index of the function in the
%   function suite.

%   If funcNum is not an integer between 1 and 15, then the user is
%   prompted for an equation and domain.

switch funcNum
    case 1
        f='2^x';
        a = 0;
        b=1;

    case 2
        f='1/x';
        a = 1;
        b = 2;
```

```
    case 3
         f='sqrt(x)';
        a = 1;
        b = 2;

    case 4
        f='1/sqrt(x)';
        a = 1;
        b = 2;

    case 5
        f='log2(x)';
        a = 1;
        b = 2;

    case 6
        f='log(x)';
        a = 1;
        b = 2;

    case 7
        f='sin(pi*x)';
        a = 0;
        b = 0.5;

    case 8
         f='cos(pi*x)';
        a = 0;
        b = 0.5;

    case 9
        f='tan(pi*x)';
        a = 0;
        b = 0.25;

    case 10
        f='sqrt(-log(x))';
        a = 1/512;
        b = 1/4;

    case 11
        f='(tan(pi*x))^2+1';
        a = 0;
        b = 0.25;

    case 12
        f='0-(x*log2(x)+(1-x)*log2(1-x))';
        a = 1/256;
        b = 1-1/256;

    case 13
        f='1/(1+exp(-x))';
        a = 0;
```

```
        b = 1;

    case 14
        f='1/(sqrt(2*pi))*exp(-x^2/2)';
        a = 0;
        b = sqrt(2);

    case 15
        f='sin(exp(x))';
        a = 0;
        b = 2;


    otherwise
        f=input(' Enter function string (ie ''e^x''): ' );
        a = input(' Enter beginning of interval: ');
        b = input(' Enter end of interval: ');

end
```

FILE: HUandDelay.m

```
function [SUP MUP BUP delay] = HUandDelay(n,device,WordWidth)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HUandDelay.m                                                                  %
%                                                                               %
% This function returns Hardware utilization parameters and propagation        %
% delay estimations for several arithmetic logic devices for a given word       %
% size n.  This does not always return the best case circuit design,
% but illustrates the effects of word-width on the size and delay of
% basic arithmetic logic circuits.
%
%   function [SUP MUP BUP delay] = HUandDelay(n,device)
%
%   Input:        n:        the wordsize of the arithmetic device
%             device:       string value for the type of logic device.  It
%                           may be one of the following devices:
%                      1. 'Adder' for an adder
%                      2. 'Mult' for multiplier built from CLBs
%                      3. 'MULT18x18' for a multiplier using MULT18x18s
%                      3. 'MUX' or 'mux' for a multiplexer
%                      4. 'RAM', 'ROM','DistRAM' for memory devices
%                      5. 'CLB' for general n-input logic function
%                      6. 'BRAM' or 'BlockRAM' for Block RAM memory
%                      7. 'BS' or 'BarrelShifter' for a BarrelShifter
%                      8. 'SIE' for a segment index encoder (LUT Cascade)
%                      9. 'MEM' or 'Mem' picks the best from ROM or BRAM
%                     10. 'SOP' for a worst case SOP with n variables
%           WordWidth:      the number of bits of the output from
%                           (used for MEM BRAM and CLB only)
```

```
%
%  Output:       SUP:   Slice Utilization Percentage
%                MUP:   MULT18x18 Utilization Percentage
%                BUP:   BRAM Utilization Percentage
%                delay: propagation delay forthe logic device
%                                                                    %
%  Comments:                                  %
%                                                                    %
% Created by:  Tim Knudstrup                                         %
%       Date:  13 October 2007                                      %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% loads the Hardware Specifications for the Xilinx Virtex-II XC2V6000
LoadXilinxDeviceData;
WordWidth= ceil(WordWidth);
%************ CALCULATING AREA USED ******************

switch device   %DistRAM assumes SinglePort (Dual Port is twice as much
space)
    case {'CLB','ROM','DistRAM','Rom','LUT'}

        % ROMs are constructed from Xilinx Primitive RAMs, using read time
        % delays from the Address input bits to the Data output bit.
        % Maximimum distributed RAM primitive is 128x1, or 7 address bits.
        % Thus, if n > 7, larger ROMs are constructed using 2^(n-7) 128x1
        % ROMs, combined with 2^(n-7):1 MUX network.  For large n > 14,
        % Block RAM should be used to avoiding using up all of the CLBs.

        fanout=ceil(2^(n-4))*WordWidth;  % also accounts for the fanout
inside each ROM unit
        if fanout > 129
            fanout =129;
        elseif fanout < 1
            fanout =1;
        end

        RomPrim=n; % m is index into a single nx1 ROM where n is at most 7
        if RomPrim > 7
            RomPrim=7;
        end
        if n > 0
            ROMdelay=tNx1ROM(RomPrim); % delay of a single Nx1 ROM (where n
<=7)
        else
            ROMdelay=0;
        end

        NumMuxLevels=n-7;
        if NumMuxLevels < 0
            NumMuxLevels = 0;
        end
        NumROMs=2^NumMuxLevels;
```

```
        LUTSperROM=ceil(2^(RomPrim-4));
        SUPperROM=100*LUTSperROM/2/TotalSlices;

        [SUP_MUX MUP_MUX BUP_MUX tMUX] =
HUandDelay(NumROMs,'MUX',WordWidth);

        if NumMuxLevels == 0
            tMUX = 0;
            SUP_MUX=0;
        end

        SUP=(NumROMs*SUPperROM+SUP_MUX)*WordWidth;
        BUP=0;
        MUP=0;
        delay = tNET(fanout)+ROMdelay+tNET(1)+tMUX;

        if n<=0
            SUP=0;
            delay=0;
        end

    case {'BlockRAM','BRAM'}

        k=ceil(n);    % k is defined in thesis as the number of address
lines
        NumMemLocations = 2^k;
        ReqMemBits = NumMemLocations*WordWidth;

        NumBlocks=ceil(ReqMemBits/MemBitsPerBRAM);
        fanout = NumBlocks;
        if fanout>128
            fanout = 128;
        end
        if fanout < 1
            fanout =1;
        end

        MuxLevels=k-14;
        if MuxLevels <= 0
            MuxLevels=0;
            SUP=0;
            MuxDelay=0;
        else
            [SUP_MUX MUP_MUX BUP_MUX MuxDelay] =
HUandDelay(2^MuxLevels,'MUX',WordWidth);
            SUP=SUP_MUX*WordWidth;
        end

        MUP=0;
        BUP=100*NumBlocks/NumBlockRAM;
        delay = tNET(fanout) + tBCKO + MuxDelay; % clk-->data out plus
Setup time
```

```
    case {'MEM', 'Mem'}
%        Uses the type of memory that requires the least hardware (HUP)

        [SUP_BRAM MUP_BRAM BUP_BRAM tBRAM] =
HUandDelay(n,'BRAM',WordWidth);
        HUP_BRAM=HUP(SUP_BRAM,MUP_BRAM,BUP_BRAM);
        [SUP_LUT MUP_LUT BUP_LUT tLUT] = HUandDelay(n,'LUT',WordWidth);
        HUP_LUT=HUP(SUP_LUT,MUP_LUT,BUP_LUT);

        if (HUP_LUT  > HUP_BRAM)
            BUP=BUP_BRAM;
            MUP=0;
            SUP=SUP_BRAM;
            delay=tBRAM;
        else
            BUP=BUP_LUT;
            MUP=0;
            SUP=SUP_LUT;
            delay=tLUT;
        end


    case 'ExtRAM' % NOT CONFIGURED AT THIS TIME
        % use Address Decoder NumLUTs
        DeviceCLBs= xxx;
        delay = xxx;
    case 'SOP'
        % This assumes a worst case SOP realization
        numTerms = 2^(n-1)*WordWidth;
        termSize=n;
        fanout=WordWidth*2^(n-1);
        if fanout>128
            fanout = 128;
        end
        if fanout < 1
            fanot = 1;
        end
        numSlices = numTerms*ceil(termSize/4)/2;
        SUP = 100*numSlices/TotalSlices;
        BUP=0;
        MUP=0;
        delay = tNET(fanout)+tLUT4+tMUXCY_S_O+(ceil(termSize/4)-
1)*tMUXCY_I_O+(numTerms)*tORCY;

    case 'Mult18x18'
        % Imported Data removes I/O Buffer gate delay, but leaves in tNET
        % Estimates for mulitpliers are from empirical data.
        maxRadix=17; % r is the radix of the multiplier
        nOVERr=ceil(n/maxRadix);
        numPPbits=ceil(n/nOVERr);  % This finds the number of bits of the
PPs
        PPGoutputBit=numPPbits*2;  % This is index into multiplier delays
for a
                                   % given pin on the MULT18x18, which is
```

```
                                % twice the number of bits in the
                                % multiplicands into the MULT18x18.

        mult = importdata('MultDelayWithNet.txt');
        [MULTn MULTt] = fillLin(mult(:,1),mult(:,2));

        fanout=nOVERr;
        NumMults=nOVERr^2;

        mult = importdata('MultSlices.txt');
        [MULTn MULTslice] = fillLin(mult(:,1),mult(:,2));
        NumSlices=MULTslice(n);

        SUP=100*NumSlices/TotalSlices;
        MUP=100*NumMults/Num18x18;
        BUP=0;
        delay= MULTt(n);
    case 'Mult'
        % Estimations based on architecture using CLBs
        Radix=4;
        nOVERr=ceil(n/Radix);
        %SlicesPerPPG=4;  % This assumes PPGs 8 4-input LUTs are used for
each PPG

        fanout=nOVERr;
        NumPPGs=nOVERr^2;
        NumAdders = 2*(nOVERr-1)*nOVERr+1;
        AdderDepth = 2*(nOVERr-1);

        % Assumes each PPG is built from a Radix-bit function
        [SUPperPPG MUP_PPG BUP_PPG PPGdelay] =
HUandDelay(Radix,'CLB',WordWidth);
        SUPperPPG = SUPperPPG * 2*Radix;  % Each PPG requires 2*Radix
functions

        % Each Adder is assumed to be a Radix-bit adder
        [SUPperAdder MUP_Adder BUP_Adder AdderDelay] =
HUandDelay(Radix,'Adder',WordWidth);

        SUP=NumPPGs*SUPperPPG+SUPperAdder*NumAdders;
        MUP=0;
        BUP=0;
        %Adders are assumed to occur in series (NOT the best design)
        delay= PPGdelay+AdderDelay*AdderDepth;

    case 'Adder'
        % Imported Data is not utilized for adders since a linear eq. fits
        % can be shown imperically from Xilinx ISE  data
        NumSlices=ceil(n/2);
        tRCA_overhead=2.528;

        % after analyzing XILINX ISE data, linear equation works for n>4
        % error in linear approximation is 0 for n > 4
```

```
        if n <= 2
            delay = tILO+tNET(1);
        elseif n <= 3
            delay = tIFX+tNET(1);
        elseif n <= 4
            delay = 2*tIF5+tNET(1);
        else
            delay = tMUXCY_I_O*(n-2) + tRCA_overhead;
        end

        SUP= 100*NumSlices/TotalSlices;
        MUP=0;
        BUP=0;


    case {'BS','BarrelShifter'}
        % uses n n:1 Muxs as most basic Barrel Shifter
        [SUP_MUX MUP_MUX BUP_MUX MuxDelay] = HUandDelay(n,'MUX',WordWidth);
        fanout = n;
        shiftLevels=ceil(log2(n));
        SUP=shiftLevels*SUP_MUX;
        MUP=0;
        BUP=0;
        delay = MuxDelay+tNET(fanout)-tNET(1);
        % removes tNET for fanout of 1 and inserts tNET for appropriate
fanout

    case {'MUX','mux','Mux'}
        % This is a n:1 MUX
        NumSlices=ceil(n/4);  % checks with ISE data

        mux = importdata('MuxDelayWithNet.txt');
        [MUXn MUXt] = fillLin(mux(:,1),mux(:,2));

        % Imported Data removes I/O Buffer gate delay, but leaves in tNET
        % Max n to index into MUXt is 128
        if n <= 128
            delay = MUXt(n);    % delay comes from imported ISE data
        else
            delay =  2*ceil(log2(n))-14+12.1997;  % estimate from equations
        end
        if n<=2
            delay=tNET(1)+tILO;
        end

        SUP=100*NumSlices/TotalSlices;
        MUP=0;
        BUP=0;

    case 'SIE'
        % SIE is assumed to be for NON-UNIFORM Segmentation
        % The SIE is constructed with a LUT cascade architecture.
        % The timing a HW utilization is based on the architecural
        % description described in the thesis with the number address lines
```

```
        % input to the memory is the WordWidth.

        k=WordWidth;
        numRails= k;
        [SUP_LUT MUP_LUT BUP_LUT LUTDelay] =
HUandDelay(k+2,'LUT',WordWidth);
        % EACH LUT is a (k+2)input LUT with k outputs --> k (k+2)input LUTs
        % are used in series.  The HUP using LUTs is compared to the HUP
        % using BRAMs and the one using less hardware is chosen.
        [SUP_BRAM MUP_BRAM BUP_BRAM BRAMDelay] =
HUandDelay(k+2,'BRAM',WordWidth);
        HUP_LUT=HUP(SUP_LUT,MUP_LUT,BUP_LUT);
        HUP_BRAM=HUP(SUP_BRAM,MUP_BRAM,BUP_BRAM);

        if HUP_LUT > HUP_BRAM
            SUP=SUP_BRAM;
            BUP=BUP_BRAM;
            MUP=MUP_BRAM;
        else
            SUP=SUP_LUT;
            BUP=BUP_LUT;
            MUP=MUP_LUT;
        end

        SUP=SUP*ceil((n-k)/2);
        BUP=BUP*ceil((n-k)/2);
        MUP=MUP*ceil((n-k)/2);

        delay=LUTDelay*ceil((n-k)/2);

    otherwise
        SUP = 'ERROR';
        BUP = 'ERROR';
        MUP = 'ERROR';
        delay = 'ERROR';
end
```

FILE: HUP.m

```
function [HUPout] = HUP(SUP,MUP,BUP)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HUP.m                                                                          %
%                                                                                %
% This function calculates the Hardware utilization percentage                   %
%                                                                                %
%    function [totHUP totalDelay] = HUP(SUP,MUP,BUP)                             %
%                                                                                %
%   Input:   SUP:       slice utilization percentage in %, max 100%              %
%            MUP:       MULT18x18 Utiliazation Percentage, max 100%              %
%            BUP:       BRAM Utilization Percentage, max 100%                    %
%                                                                                %
%  Output:   HUPout:    Calculated value for HUP.                                %
%                                                                                %
```

```
% Created by:  Tim Knudstrup                                                  %
%       Date:  12 September 2007                                             %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



BS=0;
if BS == 1

x=[1:1:100];
SUP=[1:1:100]/100;
MUP=[[1:2:100] [1:2:100]]/100;
BUP=[[1:4:100] [1:4:100] [1:4:100] [1:4:100]]/100;

HUPa = 1-((1-SUP).*(1-MUP).*(1-BUP)).^(1/3);%./sqrt((1-SUP).*(1-MUP).*(1-BUP));
HUPb=(SUP.*MUP.*BUP).^(1/3);
close all;
plot(x,SUP,x,MUP,x,BUP,x,HUPa,x,HUPb)
legend('SUP','MUP','BUP','HUPa','HUPb')

AXIS([0 100 0 1])
end

if SUP > 100
    SUP = 100;
end
if MUP > 100
    MUP = 100;
end
if BUP > 100
    BUP = 100;
end

HUPout=100*(1-((1-SUP/100)*abs(1-MUP/100)*abs(1-BUP/100))^(1/3));
```

FILE: HUPBoxes.m

```
function [totHUP totalDelay] = HUPBoxes(components,dependence,compNames)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HUPBoxes.m                                                                 %
%                                                                            %
% This function/program displays the delay and percent hardware              %
% utilization given up to 12 components and a dependence relationship.       %
% It is used to show circuit components in series and in parallel            %
% and the combined delay of multiple components which is dependent on        %
% one components relationship to another.                                    %
%                                                                            %
% function [totHUP totalDelay] = depBoxes(components,dependence,compNames)%
%                                                                            %
%   Input: components:   nx2 array of components arranged                    %
%                        n = row number = the component number               %
%                        Max number of ROWs is 12                            %
```

```
%                        each row contains :                                %
%                        [ HUP timedelay ]                                  %
%                                                                           %
%          dependence:   an nxn array that defines the dependence           %
%                        of the components.                                 %
%                        For each row, the array should contain a 1 if      %
%                        the component number (row#) has to wait until      %
%                        another component is completed (in series).        %
%                                                                           %
%          compNames:    an nx1 column of strings, naming each component    %
%                        strings must be the same length, can add extra     %
%                        spaces.                                            %
%                                                                           %
%  Output:   totHUP:    total percent of hardware used in this circuit      %
%            totalDelay: total composite circuit delay                      %
%  Comments:                                                                %
%                                                                           %
% Created by:  Tim Knudstrup                                                %
%       Date:  12 September 2007                                            %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

numComps=size(components);
numComps=numComps(1);

close all;

% Color list (each Row contains a different color code (upto 12))
Clist = [ 0.5 0 0
          0   0 0.5
          0 0.5 0
          0.5 0.5 0
          0.5 0 0.5
          0 0.5 0.5
          0.75 0 0
          0  0 0.75
          0 0.75 0
          0.75 0.75 0
          0.75 0 0.75
          0 0.75 0.75];

compEnds=zeros(1,numComps);
compStarts=compEnds;

compTop=compEnds;
compBot=compEnds;

for comp=1:numComps
    if (sum(dependence(comp,:))==0)
        compStarts(comp)=0;
    else
        compDep=find(dependence(comp,:));
        compStarts(comp)=max(compEnds(compDep));
    end
```

```
    compEnds(comp)=compStarts(comp)+components(comp,2);
end
compStarts;
compEnds;

for comp = 1:numComps
    if (comp==1)
        compBot(comp)=0;
    else
        sameStart=find(compStarts(1:comp-1)==compStarts(comp));
        if isempty(sameStart)
            compDep=find(dependence(comp,:));
            [y indx] = max(compEnds(compDep)); % finds index into
            compBot(comp)=compBot(indx);
        else
            largestTop=max(sameStart);
            compBot(comp)=compTop(largestTop);
        end
    end
    compTop(comp)=compBot(comp)+components(comp,1);
end
compBot;
compTop;

% OUTPUT Data
totalDelay=max(compEnds);
totHUP=sum(components(:,1));

% Graphs
for comp = 1:numComps
    xVals=[compStarts(comp) compStarts(comp) compEnds(comp) compEnds(comp)];
    yVals=[compBot(comp) compTop(comp) compTop(comp) compBot(comp)];

    colorset=Clist(comp,:);
    fill(xVals,yVals,colorset)
    hold on
end

legend(compNames,'Location','EastOutside')
ylabel(' Hardware Utilization Percentage')
xlabel('Propagation delay (ns)')

temp=cat(2,'Total HUP = ',num2str(totHUP,4),'%, Total Propagation Delay =
',num2str(totalDelay,4),' ns.');
title(temp)
```

FILE: LoadXilinxDeviceData.m

```
%************ XILINX Virtex-II 6000 Limits *************
% Most data originates from Virtex-II Platform FPGA Datasheet (available at
% www.xlininx.com) assuming a Virtex-II XC2V6000 device with a speed grade of
```

```
% -4 (worst case).
% Data collected through simulation is noted.
% All Delay data included here is the worst case input to output signal
% delay for the particular device.

xxx=123456789; % This value is not know at this time

%******** Available Memory  **************

% ***** Distributed SelectRAM ***
% I am really only concerned with ROM
TotalDistRAM = 132000;
TotalDistRAMbits = 1081344;
TotalDistRAMbytes = TotalDistRAMbits/8;

t_AS = 0.5;
DistRAMDelay = xxx ; % in ns
tSHCKO16 = 2.05;
tSHCKO32 = 2.49;
tSHCKOF5 = 2.23;

% ***** Block SelectRAM *********
NumBlockRAM = 144;
TotalBlockRAM = 324000;
TotalBlockRAMbits = 2654208;
TotalBlockRAMbytes = TotalBlockRAMbits/8;
MemBitsPerBRAM = 16384;

BlockRAMdelay = 2.65; % in ns
tBCKO = 2.65;
tBACK = 0.36;

% ******* ROM ******************
% Uses CLB directly as a function of n inputs
% Thus all data is imperically determined from Xilinx ISE Primitives and
% does not include net delays or IO Buffer delays.
% The delays are combinational from along the longest delay path from
% Address bit A0 to the data output.  All times are in ns.
% The primitives are actually RAM units, but only used as ROMs.
% These values do not include NET delays, they must be accounted for
% elsewhere
tNx1ROM=[0.875 0.875 0.875 0.875 0.875 0.875 1.562 1.879];
t16x1ROM=0.875;
t32x1ROM=0.875;
t64x1ROM=1.562;
t128x1ROM=1.879;
%************************************

%******* Available Logic **************
TotalSlices=33792;
TotalLUTs=67584;
TotalFFs=67584;
TotalShiftRegBits=TotalDistRAMbits;
MaxSOPChain=192;
```

```
MaxCarryChain=176;

%****** CLB *************************
TotalCLBs=TotalLUTs/8;
CLBdelay4to1 = 0.44; % SPEED GRADE -4 in ns
CLBdelay5to1 = 0.72;
tILO=0.44;
tIF5=0.72;
tIFX=0.95;
tINFXY=0.45;
tINAFX=0.32;
tINBFX=0.32;
tSOPSOP=0.44;
% MORE DATA AVAILABLE
%*************************************

%****** Multipliers ******************
% Check to see if Enhanced or not !!!
Num18x18=144;
% These are the worst case in to out delays using the entire multiplier
Delay18x18=10.36;  % in ns
Delay18x18Enh=5.91; %

% The DELAY can be reduced if the entire 18x18 Mult is not used
% See Page 22 of Module 3 in Xilinx DataSheet
% Index into the array is offset by 1
tMULT = [3.12; 3.32;3.53;3.74;3.94;4.15;4.36;4.56;
    4.77;4.98;5.19;5.39;5.6;5.81;6.01;6.22;6.43;6.63;
    6.84;7.05;7.26;7.46;7.67;7.88;8.08;8.29;8.5;8.7;
    8.91;9.12;9.33;9.53;9.74;9.95;10.15;10.36];

%*************************************

%****** Routing Delays *****************

tIBUF=0.825;
tOBUF=4.361;


%*************************************

%****** I/O Pads ***********************
TotalIOpads=1104;
IOpadDelay= 100; % in ns
%*************************************

%*************************************
% EMPIRICAL DATA COLLECTED
% This creates an array tNET from empirical data supported by Xilinx
% Datasheets.

% ***** NET DELAYS ********
data_in = importdata('NetDelay.txt');
```

```
[fanout tNET] = fillLin(data_in(:,1),data_in(:,2));

% **** SPECIAL MUX DELAYS  **
tMUXCY_I_O = 0.053; % fast carry MUX prop delay from input I0 to output O.
                    % This data is reported to be 0.05 ns in Datasheet.
tMUXCY_S_O = 0.298;

% **** LOGIC COMPONENT DELAYS
tLUT4 = 0.439;
tORCY = 0.44;

%************************************

%*****************************************************

plot_on=0;

if plot_on == 1
    stem(data_in(:,1),data_in(:,2),'bo-')
    hold on
    plot(fanout,tNET,'g.-')
    xlabel('fanout')
    ylabel('Net Delay')
    legend('Collected Data Points','FillLine Data Points');
end
```

FILE: model_Linear_NonUniform_Basic.m

```
function [totHUP totDelay] = model_Linear_NonUniform_Basic(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Linear_NonUniform_Basic.m                                               %
%                                                                               %
% This function produces the HUP and delay for a model of a linear NFG          %
%   using nonuniform segmentation.                                              %
%                                                                               %
% function [totHUP totDelay] = model_Linear_NonUniform_Basic(n,numSegs)         %
%                                                                               %
%   Input:            n:      number of bits in the system                      %
%                                                                               %
%               numSegs:      number of segments in the memory                  %
%                                                                               %
%  Output:        totHUP:    hardware utilization percentage                   %
%            totalDelay:    total composite circuit delay                      %
%  Comments:                                                                    %
%                                                                               %
% Created by:  Tim Knudstrup                                                    %
%       Date:  25 September 2007                                                %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
WordWidth=2*n;          % 2 n-bit numbers are stored in the Coefficients Memory
```

```
[SUP_SIE MUP_SIE BUP_SIE tSIE] = HUandDelay(n,'SIE',k);
[SUP_mult MUP_mult BUP_mult tMult] = HUandDelay(n,'Mult18x18',WordWidth);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(2*n,'Adder',WordWidth);

HUP_SIE= HUP(SUP_SIE, MUP_SIE, BUP_SIE);
HUP_mult= HUP(SUP_mult,MUP_mult,BUP_mult);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_SIE tSIE];
device2 = [HUP_mem tMem];
device3 = [HUP_mult tMult];
device4 = [HUP_add tAdd];

dependency= [0 0 0 0
             1 0 0 0
             0 1 0 0
             0 0 1 0];

components = [device1;device2;device3;device4];
compNames = [ 'SIE        '
              'Memory     '
              'Multiplier '
              'Adder      '];


graphON=0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Linear_Uniform_Basic.m

```
function [totHUP totDelay] = model_Linear_Uniform_Basic(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Linear_Uniform_Basic.m                                                   %
%                                                                                %
% This function produces the HUP and delay for a model of a linear NFG    %
%   using uniform segmentation.                                                  %
%                                                                                %
% function [totHUP totDelay] = model_Linear_Uniform_Basic(n,numSegs)      %
%                                                                                %
%   Input:            n:      number of bits in the system                     %
%                                                                                %
%               numSegs:      number of segments in the memory                 %
%                                                                                %
%  Output:       totHUP:    hardware utilization percentage                    %
%           totalDelay:    total composite circuit delay                       %
```

```
%  Comments:                                                                  %
%                                                                             %
% Created by:  Tim Knudstrup                                                  %
%       Date:  25 September 2007                                              %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
WordWidth=2*n;          % 2 n-bit numbers are stored in the Coefficients Memory

[SUP_mult MUP_mult BUP_mult tMult] = HUandDelay(n,'Mult18x18',WordWidth);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(2*n,'Adder',WordWidth);

HUP_mult= HUP(SUP_mult,MUP_mult,BUP_mult);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_mem tMem];
device2 = [HUP_mult tMult];
device3 = [HUP_add tAdd];

dependency= [0 0 0
             1 0 0
             0 1 0];
components = [device1;device2;device3];
compNames = [ 'Memory     '
              'Multiplier '
              'Adder      '];


graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Linear_NonUniform_Compact.m

```
function [totHUP totDelay] = model_Linear_NonUniform_Compact(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Linear_NonUniform_Compact.m                                           %
%                                                                             %
% This function produces the HUP and delay for a model of a compact           %
% linear NFG using nonuniform segmentation.                                   %
%                                                                             %
% function [totHUP totDelay] = model_Linear_NonUniform_Compact(n,numSegs)     %
```

```
%                                                                       %
%   Input:            n:      number of bits in the system              %
%                                                                       %
%              numSegs:      number of segments in the memory           %
%                                                                       %
%  Output:       totHUP:    hardware utilization percentage            %
%           totalDelay:    total composite circuit delay               %
%  Comments:                                                            %
%                                                                       %
% Created by:  Tim Knudstrup                                           %
%       Date:  25 September 2007                                        %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
q=n/2;                  % This is just an assummed value.
WordWidth=3*n-q;        % bits per word in Coefficients Memory.


[SUP_SIE MUP_SIE BUP_SIE tSIE] = HUandDelay(n,'SIE',k);
[SUP_mult MUP_mult BUP_mult tMult] =
HUandDelay(ceil(n/2),'Mult18x18',WordWidth);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(n,'Adder',WordWidth);

HUP_SIE= HUP(SUP_SIE, MUP_SIE, BUP_SIE);
HUP_mult= HUP(SUP_mult,MUP_mult,BUP_mult);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_SIE tSIE];
device2 = [HUP_mem tMem];
device3 = [HUP_add tAdd];
device4 = [HUP_mult tMult];
device5 = [HUP_add tAdd];

dependency= [0 0 0 0 0
             1 0 0 0 0
             0 1 0 0 0
             0 1 1 0 0
             0 1 0 1 0];

components = [device1;device2;device3;device4;device5];
compNames = [ 'SIE        '
              'Memory     '
              'Adder1     '
              'Multiplier '
              'Adder2     '];


graphON = 0;
if graphON == 1;
```

```
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Linear_Uniform_Compact.m

```
function [totHUP totDelay] = model_Linear_Uniform_Compact(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Linear_Uniform_Compact.m                                              %
%                                                                             %
% This function produces the HUP and delay for a compact model of a           %
% linear NFG using uniform segmentation.                                      %
%                                                                             %
% function [totHUP totDelay] = model_Linear_Uniform_Compact(n,numSegs)        %
%                                                                             %
%   Input:            n:      number of bits in the system                    %
%                                                                             %
%               numSegs:      number of segments in the memory                %
%                                                                             %
%  Output:       totHUP:    hardware utilization percentage                   %
%           totalDelay:    total composite circuit delay                     %
%  Comments:                                                                  %
%                                                                             %
% Created by:  Tim Knudstrup                                                  %
%       Date:  25 September 2007                                              %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
WordWidth=k+n;          % 2 n-bit numbers are stored in the Coefficients Memory

[SUP_mult MUP_mult BUP_mult tMult] =
HUandDelay(ceil(n/2),'Mult18x18',WordWidth);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(n,'Adder',WordWidth);

HUP_mult= HUP(SUP_mult,MUP_mult,BUP_mult);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_mem tMem];
device2 = [HUP_mult tMult];
device3 = [HUP_add tAdd];

dependency= [0 0 0
             1 0 0
             1 1 0];
components = [device1;device2;device3];
compNames = [ 'Memory     '
              'Multiplier '
              'Adder      '];
```

```
graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Quad_NonUniform_Basic.m

```
function [totHUP totDelay] = model_Quad_NonUniform_Basic(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Quad_NonUniform_Basic.m                                                   %
%                                                                                 %
% This function produces the HUP and delay for a model of a quadratic NFG %
%   using nonuniform segmentation.                                                %
%                                                                                 %
% function [totHUP totDelay] = model_Quad_NonUniform_Basic(n,numSegs)     %
%                                                                                 %
%   Input:            n:      number of bits in the system                      %
%                                                                                 %
%               numSegs:      number of segments in the memory                  %
%                                                                                 %
%  Output:       totHUP:    hardware utilization percentage                     %
%            totalDelay:    total composite circuit delay                       %
%  Comments:                                                                      %
%                                                                                 %
% Created by:  Tim Knudstrup                                                      %
%       Date:  25 September 2007                                                  %
%                                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
WordWidth=3*n;          % 3 n-bit numbers are stored in the Coefficients Memory


[SUP_SIE MUP_SIE BUP_SIE tSIE] = HUandDelay(n,'SIE',k);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_mult_2N MUP_mult_2N BUP_mult_2N tMult_2N] =
HUandDelay(n,'Mult18x18',WordWidth);
[SUP_mult_3N MUP_mult_3N BUP_mult_3N tMult_3N] =
HUandDelay(ceil(1.5*n),'Mult18x18',WordWidth);
[SUP_add_2N MUP_add_2N BUP_add_2N tAdd_2N] = HUandDelay(2*n,'Adder',WordWidth);
[SUP_add_3N MUP_add_3N BUP_add_3N tAdd_3N] = HUandDelay(3*n,'Adder',WordWidth);

HUP_SIE= HUP(SUP_SIE, MUP_SIE, BUP_SIE);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);

HUP_mult_2N = HUP(SUP_mult_2N,MUP_mult_2N,BUP_mult_2N);
HUP_mult_3N = HUP(SUP_mult_3N,MUP_mult_3N,BUP_mult_3N);
```

```
HUP_add_2N= HUP(SUP_add_2N, MUP_add_2N, BUP_add_2N);
HUP_add_3N= HUP(SUP_add_3N, MUP_add_3N, BUP_add_3N);

device1 = [HUP_SIE tSIE];
device2 = [HUP_mem tMem];
device3 = [HUP_mult_2N tMult_2N];
device4 = [HUP_mult_2N tMult_2N];
device5 = [HUP_mult_3N tMult_3N];
device6 = [HUP_add_2N tAdd_2N];
device7 = [HUP_add_3N tAdd_3N];

dependency= [0 0 0 0 0 0 0
             1 0 0 0 0 0 0
             0 0 0 0 0 0 0
             0 1 0 0 0 0 0
             0 1 1 0 0 0 0
             0 1 0 1 0 0 0
             0 0 0 0 1 1 0  ];

components = [device1;device2;device3;device4;device5; device6; device7];
compNames = [ 'SIE          '
              'Coeff. Table '
              'Multiplier 1 '
              'Multiplier 2 '
              'Multiplier 3 '
              'Adder 1      '
              'Adder 2      '];


graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Quad_Uniform_Basic.m

```
function [totHUP totDelay] = model_Quad_Uniform_Basic(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Quad_Uniform_Basic.m                                                    %
%                                                                               %
% This function produces the HUP and delay for a basic model of a               %
%   quadratic NFG using uniform segmentation.                                   %
%                                                                               %
% function [totHUP totDelay] = model_Quad_Uniform_Basic(n,numSegs)              %
%                                                                               %
%   Input:            n:      number of bits in the system                      %
%                                                                               %
%              numSegs:      number of segments in the memory                  %
%                                                                               %
```

```
%  Output:       totHUP:    hardware utilization percentage                %
%            totalDelay:    total composite circuit delay                  %
%  Comments:                                                               %
%                                                                          %
% Created by:  Tim Knudstrup                                               %
%       Date:  25 September 2007                                           %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
WordWidth=3*n;          % 3 n-bit numbers are stored in the Coefficients Memory


%[SUP_SIE MUP_SIE BUP_SIE tSIE] = HUandDelay(n,'SIE',k);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_mult_2N MUP_mult_2N BUP_mult_2N tMult_2N] =
HUandDelay(n,'Mult18x18',WordWidth);
[SUP_mult_3N MUP_mult_3N BUP_mult_3N tMult_3N] =
HUandDelay(ceil(1.5*n),'Mult18x18',WordWidth);
[SUP_add_2N MUP_add_2N BUP_add_2N tAdd_2N] = HUandDelay(2*n,'Adder',WordWidth);
[SUP_add_3N MUP_add_3N BUP_add_3N tAdd_3N] = HUandDelay(3*n,'Adder',WordWidth);

%HUP_SIE= HUP(SUP_SIE, MUP_SIE, BUP_SIE);
HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);

HUP_mult_2N = HUP(SUP_mult_2N,MUP_mult_2N,BUP_mult_2N);
HUP_mult_3N = HUP(SUP_mult_3N,MUP_mult_3N,BUP_mult_3N);

HUP_add_2N= HUP(SUP_add_2N, MUP_add_2N, BUP_add_2N);
HUP_add_3N= HUP(SUP_add_3N, MUP_add_3N, BUP_add_3N);

%device1 = [HUP_SIE tSIE];
device1 = [HUP_mem tMem];
device2 = [HUP_mult_2N tMult_2N];
device3 = [HUP_mult_2N tMult_2N];
device4 = [HUP_mult_3N tMult_3N];
device5 = [HUP_add_2N tAdd_2N];
device6 = [HUP_add_3N tAdd_3N];

dependency= [0 0 0 0 0 0
             0 0 0 0 0 0
             1 0 0 0 0 0
             1 1 0 0 0 0
             1 0 1 0 0 0
             0 0 0 1 1 0   ];

components = [device1;device2;device3;device4;device5; device6;];
compNames = [ 'Coeff. Table '
              'Multiplier 1 '
              'Multiplier 2 '
              'Multiplier 3 '
              'Adder 1      '
```

```
              'Adder 2      '];


graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Quad_NonUniform_Compact.m

```
function [totHUP totDelay] = model_Quad_NonUniform_Compact(n,numSegs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% model_Quad_NonUniform_Compact.m                                             %
%                                                                             %
% This function produces the HUP and delay for a model of a compact           %
%   quadratic NFG using nonuniform segmentation.                              %
%                                                                             %
% function [totHUP totDelay] = model_Quad_NonUniform_Compact(n,numSegs)       %
%                                                                             %
%   Input:             n:      number of bits in the system                   %
%                                                                             %
%                 numSegs:      number of segments in the memory               %
%                                                                             %
%  Output:          totHUP:    hardware utilization percentage                %
%              totalDelay:     total composite circuit delay                  %
%  Comments:                                                                  %
%                                                                             %
% Created by:  Tim Knudstrup                                                  %
%       Date:  25 September 2007                                              %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
q1=n/2;                 % these are just example q's
q2=n/2;
WordWidth=4*n-q1-q2;          %  Coefficients Memory


[SUP_SIE MUP_SIE BUP_SIE tSIE] = HUandDelay(n,'SIE',k);
[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_mult_q MUP_mult_q BUP_mult_q tMult_q] =
HUandDelay(ceil(q2/2),'Mult18x18',WordWidth);
[SUP_mult_N MUP_mult_N BUP_mult_N tMult_N] =
HUandDelay(ceil(n/2),'Mult18x18',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(n,'Adder',WordWidth);

HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
```

```
HUP_SIE= HUP(SUP_SIE, MUP_SIE, BUP_SIE);
HUP_mult_q = HUP(SUP_mult_q,MUP_mult_q,BUP_mult_q);
HUP_mult_N = HUP(SUP_mult_N,MUP_mult_N,BUP_mult_N);
HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_SIE tSIE];
device2 = [HUP_mem tMem];
device3 = [HUP_add tAdd];
device4 = [HUP_mult_q tMult_q];
device5 = [HUP_mult_N tMult_N];
device6 = [HUP_mult_N tMult_N];
device7 = [HUP_add tAdd];
device8 = [HUP_add tAdd];

dependency= [0 0 0 0 0 0 0 0
             1 0 0 0 0 0 0 0
             0 1 0 0 0 0 0 0
             0 0 1 0 0 0 0 0
             0 1 1 0 0 0 0 0
             0 1 0 1 0 0 0 0
             0 1 0 0 1 0 0 0
             0 0 0 0 0 1 1 0];

components = [device1;device2;device3;device4;device5; device6; device7;
device8];
compNames = [ 'SIE          '
              'Coeff. Table '
              'Adder 1      '
              'Multiplier 1 '
              'Multiplier 2 '
              'Multiplier 3 '
              'Adder 2      '
              'Adder 3      '];


graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: model_Quad_Uniform_Compact.m

```
function [totHUP totDelay] = model_Quad_Uniform_Compact(n,numSegs)
%%%
```

```
%                                                                       %
%   Input:            n:      number of bits in the system              %
%                                                                       %
%              numSegs:      number of segments in the memory           %
%                                                                       %
%  Output:       totHUP:    hardware utilization percentage            %
%           totalDelay:    total composite circuit delay               %
%  Comments:                                                            %
%                                                                       %
% Created by:  Tim Knudstrup                                           %
%       Date:  25 September 2007                                        %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


k=ceil(log2(numSegs));  % number of address lines to the coefficients Memory
q1=n/2;                 % these are just example q's
q2=n/2;
WordWidth=4*n-q1-q2;          % Coefficients Memory


[SUP_mem MUP_mem BUP_mem tMem] = HUandDelay(k,'MEM',WordWidth);
[SUP_mult_q MUP_mult_q BUP_mult_q tMult_q] =
HUandDelay(ceil(q2/2),'Mult18x18',WordWidth);
[SUP_mult_N MUP_mult_N BUP_mult_N tMult_N] =
HUandDelay(ceil(n/2),'Mult18x18',WordWidth);
[SUP_add MUP_add BUP_add tAdd] = HUandDelay(n,'Adder',WordWidth);

HUP_mem= HUP(SUP_mem, MUP_mem, BUP_mem);
HUP_mult_q = HUP(SUP_mult_q,MUP_mult_q,BUP_mult_q);
HUP_mult_N = HUP(SUP_mult_N,MUP_mult_N,BUP_mult_N);

HUP_add= HUP(SUP_add, MUP_add, BUP_add);

device1 = [HUP_mem tMem];
device2 = [HUP_add tAdd];
device3 = [HUP_mult_q tMult_q];
device4 = [HUP_mult_N tMult_N];
device5 = [HUP_mult_N tMult_N];
device6 = [HUP_add tAdd];
device7 = [HUP_add tAdd];

dependency= [0 0 0 0 0 0 0
             1 0 0 0 0 0 0
             0 1 0 0 0 0 0
             1 1 0 0 0 0 0
             1 0 1 0 0 0 0
             1 0 0 1 0 0 0
             0 0 0 0 1 1 0];

components = [device1;device2;device3;device4;device5; device6;device7];
compNames = [ 'Coeff. Table '
              'Adder 1      '
              'Multiplier 1 '
```

```
               'Multiplier 2 '
               'Multiplier 3 '
               'Adder 2      '
               'Adder 3      '];


graphON = 0;
if graphON == 1;
    [totHUP totDelay] = HUPBoxes(components,dependency,compNames);
else
    [totHUP totDelay] = totalHUPandDelay(components,dependency,compNames);
end
```

FILE: myInt.m

```
function [intVal]= myInt(f_symbol,a,b)

%   This function returns an approximation for the integral of the symbolic
%   function over the interval a to b.  The approximation is calculated
%   using trapezoidal integration approximation.

numPts=10000;

rez=(b-a)/numPts;
X=[a:rez:b];
y=(subs(f_symbol,X));

totSum = 0;
width= X(2)-X(1);

for ii=1:length(X)-1
    incSum= width*y(ii)+0.5*width*(y(ii+1)-y(ii));
    totSum=totSum+incSum;
end

intVal=totSum;
```

FILE: pickModel.m

```
function [totHUP totDelay] = pickModel(modelNum,n,segs);

%   This function returns the total HUP and Delay for a function
%   implemented on an NFG model chosen by 'modelNum.'
%   The default model is the basic linear NFG with uniform segmentation
%   (LUB).

switch modelNum
    case 1
        [totHUP totDelay] = model_Linear_Uniform_Basic(n,segs);
```

```
    case 2
        [totHUP totDelay] = model_Linear_NonUniform_Basic(n,segs);
    case 5
        [totHUP totDelay] = model_Linear_Uniform_Compact(n,segs);
    case 6
        [totHUP totDelay] = model_Linear_NonUniform_Compact(n,segs);
    case 3
        [totHUP totDelay] = model_Quad_Uniform_Basic(n,segs);
    case 4
        [totHUP totDelay] = model_Quad_NonUniform_Basic(n,segs);
    case 7
        [totHUP totDelay] = model_Quad_Uniform_Compact(n,segs);
    case 8
        [totHUP totDelay] = model_Quad_NonUniform_Compact(n,segs);
    otherwise
        [totHUP totDelay] = model_Linear_Uniform_Basic(n,segs);
end
```

FILE: segments.m

```
function [numSegs] = segments(f,xmin,xmax,n)
%%%
```

```
b=xmax;
epps=2^(-n-1);
f_of_x=func(x);

FirstDeriv= diff(f_of_x,'x');
SecondDeriv= diff(FirstDeriv,'x');

sqrt_2ndDeriv=sqrt((SecondDeriv));
%SegsLin = abs(0.25*int((sqrt_2ndDeriv),'x',a,b)/sqrt(epps))
numSegsLin_NONUNIFORM = ceil(0.25*myInt(abs(sqrt_2ndDeriv),a,b)/sqrt(epps));
thirdDeriv=diff(SecondDeriv,'x');
%SegsQuad = abs(0.25 * int(((thirdDeriv))^(1/3),a,b)/(3*epps)^(1/3))
numSegsQuad_NONUNIFORM =
ceil(0.25*myInt(abs(thirdDeriv)^(1/3),a,b)/(3*epps)^(1/3));

%   Substituting values
a=xmin;
b=xmax;
epps= 2^(-n-1);

%numSegsLin_NONUNIFORM=ceil(abs(subs(SegsLin)));
%numSegsQuad_NONUNIFORM=ceil(abs(subs(SegsQuad)));

dummyX=[a:(b-a)/100:b]';
max_2ndDeriv=max(abs((subs(SecondDeriv,dummyX))));
segWidth_Linear=4*sqrt(epps/max_2ndDeriv);

max_3rdDeriv=max(abs(subs(thirdDeriv,dummyX)));
segWidth_Quad=4*(3*epps/max_3rdDeriv)^(1/3);

numSegsLin_UNIFORM=ceil((b-a)/segWidth_Linear);
numSegsQuad_UNIFORM=ceil((b-a)/segWidth_Quad);

numSegs=[numSegsLin_UNIFORM;
    numSegsLin_NONUNIFORM;
    numSegsQuad_UNIFORM;
    numSegsQuad_NONUNIFORM];
```

FILE: totalHUPandDelay.m

```
function [totHUP totalDelay] = totalHUPandDelay(components,dependence,compNames)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% depBoxes.m                                                                   %
%                                                                              %
% This function/program calculates the delay and percent hardware              %
% utilization given up to 12 components and a dependence relationship.         %
% It is used to calculate circuit components in series and in parallel         %
% and the combined delay of multiple components which is dependent on          %
% one components relationship to another.                                      %
%                                                                              %
%   This function was modified from HUPboxes, which plots the outputs          %
%                                                                              %
```

```
% function [totHUP totalDelay] =
totalHUPandDelay(components,dependence,compNames)
%                                                                       %
%   Input: components:   nx2 array of components arranged               %
%                        n = row number = the component number          %
%                        Max number of ROWs is 12                       %
%                                                                       %
%                        each row contains :                            %
%                        [ HUP timedelay ]                              %
%                                                                       %
%          dependence:   an nxn array that defines the dependence       %
%                        of the components.                             %
%                        For each row, the array should contain a 1 if  %
%                        the component number (row#) has to wait until  %
%                        another component is completed (in series).    %
%                                                                       %
%          compNames:    an nx1 column of strings, naming each component %
%                        strings must be the same length, can add extra %
%                        spaces.                                        %
%                                                                       %
%  Output:   totHUP:    hardware utilization percentage                 %
%            totalDelay: total composite circuit delay                  %
%  Comments:                                                            %
%                                                                       %
% Created by:  Tim Knudstrup                                            %
%       Date:  25 September 2007                                        %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

numComps=size(components);
numComps=numComps(1);

% Color list (each Row contains a different color code (upto 12))
Clist = [ 0.5 0 0
          0   0 0.5
          0 0.5 0
          0.5 0.5 0
          0.5 0 0.5
          0 0.5 0.5
          0.75 0 0
          0  0 0.75
          0 0.75 0
          0.75 0.75 0
          0.75 0 0.75
          0 0.75 0.75];

compEnds=zeros(1,numComps);
compStarts=compEnds;

compTop=compEnds;
compBot=compEnds;

for comp=1:numComps
    if (sum(dependence(comp,:))==0)
```

```
        compStarts(comp)=0;
    else
        compDep=find(dependence(comp,:));
        compStarts(comp)=max(compEnds(compDep));
    end
    compEnds(comp)=compStarts(comp)+components(comp,2);
end
compStarts;
compEnds;

for comp = 1:numComps
    if (comp==1)
        compBot(comp)=0;
    else
        sameStart=find(compStarts(1:comp-1)==compStarts(comp));
        if isempty(sameStart)
            compDep=find(dependence(comp,:));
            [y indx] = max(compEnds(compDep)); % finds index into
            compBot(comp)=compBot(indx);
        else
            largestTop=max(sameStart);
            compBot(comp)=compTop(largestTop);
        end
    end
    compTop(comp)=compBot(comp)+components(comp,1);
end
compBot;
compTop;

% OUTPUT Data
totalDelay=max(compEnds);
totHUP=sum(components(:,1));
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. DATA COLLECTION

B.1 DATA COLLECTION WITH XILINX ISE PROJECT NAVIGATOR

Xilinx ISE Project Navigator was used extensively to construct schematic and behavioral sources in order to estimate hardware utilization and delay.

1. HDL Sources

Behavioral VHDL sources were written in Xilinx ISE Project Navigator for multipliers and adders. Some circuits were constructed from schematics using Xilinx's primitive hardware. These circuits produced verilog code during the synthesis process. The vf-files for the schematic circuits are also shown in this appendix.

The VHDL sources have been changed during the data collection phase of this thesis in order to collect information on various sized circuits. For example, the number of input and output bits of the behavioral adder were altered for various values between 1 and 129. The VHDL code shown in this appendix is the most recently used file.

FILE: Adder_64.vhd

```
----------------------------------------------------------------------------------
-- Company:        NPS
-- Engineer:       Tim Knudstrup
--
-- Create Date:    08/2/07
-- Design Name:
-- Module Name:    adder_64bit - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Adder_64 is
    Port (    a : in  std_logic_vector(128 downto 0);
              b : in  std_logic_vector(128 downto 0);
            sum : out std_logic_vector(128  downto 0));
end Adder_64;

architecture Behavioral of Adder_64 is

begin

            sum <= a+b;

end Behavioral;
```

FILE: Multiplier.vhd

```
----------------------------------------------------------------------------------
-- Company:        NPS
-- Engineer:       Tim Knudstrup
--
-- Create Date:    08/2/07
-- Design Name:
-- Module Name:    Multiplier - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Multiplier is
    Port (    a : in  std_logic_vector(16 downto 0);
              b : in  std_logic_vector(16 downto 0);
            sum : out std_logic_vector(33 downto 0));
```

```
end Multiplier;

architecture Behavioral of Multiplier is

begin
            sum <= a*b;

end Behavioral;
```

FILE: mux128to1.vf

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.
//////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 9.2.02i
//  \   \         Application : sch2verilog
//  /   /         Filename : mux128to1.vf
// /___/   /\     Timestamp : 11/11/2007 12:03:00
// \   \  /  \
//  \___\/\___\
//
//Command: C:\Xilinx92i\bin\nt\sch2verilog.exe -intstyle ise -family virtex2 -w
"C:/Documents and Settings/HP_Owner/My
Documents/schoolStuff/Thesis/VHDL/ThesisVHDLSims/mux128to1.sch" mux128to1.vf
//Design Name: mux128to1
//Device: virtex2
//Purpose:
//    This verilog netlist is translated from an ECS schematic.It can be
//    synthesized and simulated, but it should not be modified.
//
`timescale 1ns / 1ps

module M2_1E_MXILINX_mux128to1(D0,
                               D1,
                               E,
                               S0,
                               O);

    input D0;
    input D1;
    input E;
    input S0;
   output O;

   wire M0;
   wire M1;

   AND3 I_36_30 (.I0(D1),
                 .I1(E),
                 .I2(S0),
                 .O(M1));
   AND3B1 I_36_31 (.I0(S0),
```

```
                    .I1(E),
                    .I2(D0),
                    .O(M0));
   OR2 I_36_38 (.I0(M1),
                .I1(M0),
                .O(O));
endmodule
`timescale 1ns / 1ps

module M4_1E_MXILINX_mux128to1(D0,
                               D1,
                               D2,
                               D3,
                               E,
                               S0,
                               S1,
                               O);

    input D0;
    input D1;
    input D2;
    input D3;
    input E;
    input S0;
    input S1;
   output O;

   wire M01;
   wire M23;

   M2_1E_MXILINX_mux128to1 I_M01 (.D0(D0),
                                  .D1(D1),
                                  .E(E),
                                  .S0(S0),
                                  .O(M01));
   // synthesis attribute HU_SET of I_M01 is "I_M01_1"
   M2_1E_MXILINX_mux128to1 I_M23 (.D0(D2),
                                  .D1(D3),
                                  .E(E),
                                  .S0(S0),
                                  .O(M23));
   // synthesis attribute HU_SET of I_M23 is "I_M23_0"
   MUXF5 I_O (.I0(M01),
              .I1(M23),
              .S(S1),
              .O(O));
endmodule
`timescale 1ns / 1ps

module mux128to1(DataIn,
                 Sel,
                 XLXN_9,
                 XLXN_20);

    input [127:0] DataIn;
```

```
    input [6:0] Sel;
    input XLXN_9;
   output XLXN_20;

   wire XLXN_1;
   wire XLXN_2;
   wire XLXN_3;
   wire XLXN_4;

   mux32to1 XLXI_2 (.CE(XLXN_9),
                    .dataIn(DataIn[127:96]),
                    .Sel(Sel[4:0]),
                    .XLXN_125(XLXN_1));
   mux32to1 XLXI_3 (.CE(XLXN_9),
                    .dataIn(DataIn[95:64]),
                    .Sel(Sel[4:0]),
                    .XLXN_125(XLXN_2));
   mux32to1 XLXI_4 (.CE(XLXN_9),
                    .dataIn(DataIn[63:32]),
                    .Sel(Sel[4:0]),
                    .XLXN_125(XLXN_3));
   mux32to1 XLXI_5 (.CE(XLXN_9),
                    .dataIn(DataIn[31:0]),
                    .Sel(Sel[4:0]),
                    .XLXN_125(XLXN_4));
   M4_1E_MXILINX_mux128to1 XLXI_6 (.D0(XLXN_1),
                                   .D1(XLXN_2),
                                   .D2(XLXN_3),
                                   .D3(XLXN_4),
                                   .E(XLXN_9),
                                   .S0(Sel[5]),
                                   .S1(Sel[6]),
                                   .O(XLXN_20));
   // synthesis attribute HU_SET of XLXI_6 is "XLXI_6_2"
endmodule
```

FILE: fanouts.vf

```
////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.
////////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 9.2.02i
//  \   \         Application : sch2verilog
//  /   /         Filename : fanouts.vf
// /___/   /\     Timestamp : 11/11/2007 12:03:12
// \   \  /  \
//  \___\/\___\
//
//Command: C:\Xilinx92i\bin\nt\sch2verilog.exe -intstyle ise -family virtex2 -w
"C:/Documents and Settings/HP_Owner/My
Documents/schoolStuff/Thesis/VHDL/ThesisVHDLSims/fanouts.sch" fanouts.vf
//Design Name: fanouts
```

```
//Device: virtex2
//Purpose:
//    This verilog netlist is translated from an ECS schematic.It can be
//    synthesized and simulated, but it should not be modified.
//
`timescale 1ns / 1ps

module AND12_MXILINX_fanouts(I0,
                             I1,
                             I2,
                             I3,
                             I4,
                             I5,
                             I6,
                             I7,
                             I8,
                             I9,
                             I10,
                             I11,
                             O);

    input I0;
    input I1;
    input I2;
    input I3;
    input I4;
    input I5;
    input I6;
    input I7;
    input I8;
    input I9;
    input I10;
    input I11;
   output O;

   wire dummy;
   wire S0;
   wire S1;
   wire S2;
   wire O_DUMMY;

   assign O = O_DUMMY;
   FMAP I_36_29 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I3),
                 .O(S0));
   // synthesis attribute RLOC of I_36_29 is "X0Y0"
   AND4 I_36_110 (.I0(I0),
                  .I1(I1),
                  .I2(I2),
                  .I3(I3),
                  .O(S0));
   AND4 I_36_127 (.I0(I4),
                  .I1(I5),
```

```
                  .I2(I6),
                  .I3(I7),
                  .O(S1));
   FMAP I_36_138 (.I1(I4),
                  .I2(I5),
                  .I3(I6),
                  .I4(I7),
                  .O(S1));
   // synthesis attribute RLOC of I_36_138 is "X0Y0"
   FMAP I_36_142 (.I1(I8),
                  .I2(I9),
                  .I3(I10),
                  .I4(I11),
                  .O(S2));
   // synthesis attribute RLOC of I_36_142 is "X0Y1"
   AND4 I_36_151 (.I0(I8),
                  .I1(I9),
                  .I2(I10),
                  .I3(I11),
                  .O(S2));
   AND3 I_36_177 (.I0(S0),
                  .I1(S1),
                  .I2(S2),
                  .O(O_DUMMY));
   FMAP I_36_181 (.I1(S0),
                  .I2(S1),
                  .I3(S2),
                  .I4(dummy),
                  .O(O_DUMMY));
   // synthesis attribute RLOC of I_36_181 is "X0Y1"
endmodule
`timescale 1ns / 1ps

module AND16_MXILINX_fanouts(I0,
                             I1,
                             I2,
                             I3,
                             I4,
                             I5,
                             I6,
                             I7,
                             I8,
                             I9,
                             I10,
                             I11,
                             I12,
                             I13,
                             I14,
                             I15,
                             O);

    input I0;
    input I1;
    input I2;
    input I3;
```

```
    input I4;
    input I5;
    input I6;
    input I7;
    input I8;
    input I9;
    input I10;
    input I11;
    input I12;
    input I13;
    input I14;
    input I15;
   output O;

   wire CIN;
   wire C0;
   wire C1;
   wire C2;
   wire S0;
   wire S1;
   wire S2;
   wire S3;
   wire XLXN_46;

   MUXCY_L I_36_2 (.CI(CIN),
                   .DI(XLXN_46),
                   .S(S0),
                   .LO(C0));
   // synthesis attribute RLOC of I_36_2 is "X0Y0"
   FMAP I_36_29 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I3),
                 .O(S0));
   // synthesis attribute RLOC of I_36_29 is "X0Y0"
   VCC I_36_107 (.P(CIN));
   GND I_36_109 (.G(XLXN_46));
   AND4 I_36_110 (.I0(I0),
                  .I1(I1),
                  .I2(I2),
                  .I3(I3),
                  .O(S0));
   AND4 I_36_127 (.I0(I4),
                  .I1(I5),
                  .I2(I6),
                  .I3(I7),
                  .O(S1));
   MUXCY_L I_36_129 (.CI(C0),
                     .DI(XLXN_46),
                     .S(S1),
                     .LO(C1));
   // synthesis attribute RLOC of I_36_129 is "X0Y0"
   FMAP I_36_138 (.I1(I4),
                  .I2(I5),
                  .I3(I6),
```

```
                  .I4(I7),
                  .O(S1));
   // synthesis attribute RLOC of I_36_138 is "X0Y0"
   FMAP I_36_142 (.I1(I8),
                  .I2(I9),
                  .I3(I10),
                  .I4(I11),
                  .O(S2));
   // synthesis attribute RLOC of I_36_142 is "X0Y1"
   MUXCY_L I_36_147 (.CI(C1),
                     .DI(XLXN_46),
                     .S(S2),
                     .LO(C2));
   // synthesis attribute RLOC of I_36_147 is "X0Y1"
   AND4 I_36_151 (.I0(I8),
                  .I1(I9),
                  .I2(I10),
                  .I3(I11),
                  .O(S2));
   AND4 I_36_161 (.I0(I12),
                  .I1(I13),
                  .I2(I14),
                  .I3(I15),
                  .O(S3));
   MUXCY I_36_165 (.CI(C2),
                   .DI(XLXN_46),
                   .S(S3),
                   .O(O));
   // synthesis attribute RLOC of I_36_165 is "X0Y1"
   FMAP I_36_170 (.I1(I12),
                  .I2(I13),
                  .I3(I14),
                  .I4(I15),
                  .O(S3));
   // synthesis attribute RLOC of I_36_170 is "X0Y1"
endmodule
`timescale 1ns / 1ps

module AND9_MXILINX_fanouts(I0,
                            I1,
                            I2,
                            I3,
                            I4,
                            I5,
                            I6,
                            I7,
                            I8,
                            O);

    input I0;
    input I1;
    input I2;
    input I3;
    input I4;
    input I5;
```

```
    input I6;
    input I7;
    input I8;
   output O;

   wire dummy;
   wire S0;
   wire S1;
   wire O_DUMMY;

   assign O = O_DUMMY;
   FMAP I_36_29 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I3),
                 .O(S0));
   // synthesis attribute RLOC of I_36_29 is "X0Y0"
   AND4 I_36_110 (.I0(I0),
                  .I1(I1),
                  .I2(I2),
                  .I3(I3),
                  .O(S0));
   AND4 I_36_127 (.I0(I4),
                  .I1(I5),
                  .I2(I6),
                  .I3(I7),
                  .O(S1));
   FMAP I_36_138 (.I1(I4),
                  .I2(I5),
                  .I3(I6),
                  .I4(I7),
                  .O(S1));
   // synthesis attribute RLOC of I_36_138 is "X0Y0"
   FMAP I_36_142 (.I1(S0),
                  .I2(S1),
                  .I3(I8),
                  .I4(dummy),
                  .O(O_DUMMY));
   // synthesis attribute RLOC of I_36_142 is "X0Y1"
   AND3 I_36_176 (.I0(S0),
                  .I1(S1),
                  .I2(I8),
                  .O(O_DUMMY));
endmodule
`timescale 1ns / 1ps

module AND8_MXILINX_fanouts(I0,
                            I1,
                            I2,
                            I3,
                            I4,
                            I5,
                            I6,
                            I7,
                            O);
```

```
    input I0;
    input I1;
    input I2;
    input I3;
    input I4;
    input I5;
    input I6;
    input I7;
   output O;

   wire dummy;
   wire S0;
   wire S1;
   wire O_DUMMY;

   assign O = O_DUMMY;
   FMAP I_36_29 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I3),
                 .O(S0));
   // synthesis attribute RLOC of I_36_29 is "X0Y0"
   AND4 I_36_110 (.I0(I0),
                  .I1(I1),
                  .I2(I2),
                  .I3(I3),
                  .O(S0));
   AND4 I_36_127 (.I0(I4),
                  .I1(I5),
                  .I2(I6),
                  .I3(I7),
                  .O(S1));
   FMAP I_36_138 (.I1(I4),
                  .I2(I5),
                  .I3(I6),
                  .I4(I7),
                  .O(S1));
   // synthesis attribute RLOC of I_36_138 is "X0Y0"
   AND2 I_36_142 (.I0(S0),
                  .I1(S1),
                  .O(O_DUMMY));
   FMAP I_36_152 (.I1(S0),
                  .I2(S1),
                  .I3(dummy),
                  .I4(dummy),
                  .O(O_DUMMY));
   // synthesis attribute RLOC of I_36_152 is "X0Y1"
endmodule
`timescale 1ns / 1ps

module AND7_MXILINX_fanouts(I0,
                            I1,
                            I2,
                            I3,
```

```
                            I4,
                            I5,
                            I6,
                            O);

    input I0;
    input I1;
    input I2;
    input I3;
    input I4;
    input I5;
    input I6;
   output O;

   wire I36;
   wire O_DUMMY;

   assign O = O_DUMMY;
   AND4 I_36_69 (.I0(I3),
                 .I1(I4),
                 .I2(I5),
                 .I3(I6),
                 .O(I36));
   AND4 I_36_85 (.I0(I0),
                 .I1(I1),
                 .I2(I2),
                 .I3(I36),
                 .O(O_DUMMY));
   FMAP I_36_98 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I36),
                 .O(O_DUMMY));
   // synthesis attribute RLOC of I_36_98 is "X0Y0"
   FMAP I_36_110 (.I1(I3),
                  .I2(I4),
                  .I3(I5),
                  .I4(I6),
                  .O(I36));
   // synthesis attribute RLOC of I_36_110 is "X0Y0"
endmodule
`timescale 1ns / 1ps

module AND6_MXILINX_fanouts(I0,
                            I1,
                            I2,
                            I3,
                            I4,
                            I5,
                            O);

    input I0;
    input I1;
    input I2;
    input I3;
```

```
    input I4;
    input I5;
   output O;

   wire dummy;
   wire I35;
   wire O_DUMMY;

   assign O = O_DUMMY;
   AND3 I_36_69 (.I0(I3),
                 .I1(I4),
                 .I2(I5),
                 .O(I35));
   AND4 I_36_85 (.I0(I0),
                 .I1(I1),
                 .I2(I2),
                 .I3(I35),
                 .O(O_DUMMY));
   FMAP I_36_93 (.I1(I3),
                 .I2(I4),
                 .I3(I5),
                 .I4(dummy),
                 .O(I35));
   // synthesis attribute RLOC of I_36_93 is "X0Y0"
   FMAP I_36_94 (.I1(I0),
                 .I2(I1),
                 .I3(I2),
                 .I4(I35),
                 .O(O_DUMMY));
   // synthesis attribute RLOC of I_36_94 is "X0Y0"
endmodule
`timescale 1ns / 1ps

module fanouts(XLXN_115,
               XLXN_520,
               XLXN_537,
               XLXN_118,
               XLXN_144,
               XLXN_483,
               XLXN_484,
               XLXN_485,
               XLXN_486,
               XLXN_487,
               XLXN_521,
               XLXN_522,
               XLXN_523,
               XLXN_524,
               XLXN_525,
               XLXN_603);

    input XLXN_115;
    input XLXN_520;
    input XLXN_537;
   output XLXN_118;
   output XLXN_144;
```

```
    output XLXN_483;
    output XLXN_484;
    output XLXN_485;
    output XLXN_486;
    output XLXN_487;
    output XLXN_521;
    output XLXN_522;
    output XLXN_523;
    output XLXN_524;
    output XLXN_525;
    output XLXN_603;

    wire XLXN_11;
    wire XLXN_112;
    wire XLXN_152;
    wire XLXN_212;
    wire XLXN_249;
    wire XLXN_258;
    wire XLXN_503;
    wire XLXN_506;
    wire XLXN_509;
    wire XLXN_517;

    AND2 XLXI_3 (.I0(XLXN_115),
                 .I1(XLXN_115),
                 .O(XLXN_112));
    AND3 XLXI_4 (.I0(XLXN_112),
                 .I1(XLXN_112),
                 .I2(XLXN_112),
                 .O(XLXN_152));
    AND4 XLXI_5 (.I0(XLXN_152),
                 .I1(XLXN_152),
                 .I2(XLXN_152),
                 .I3(XLXN_152),
                 .O(XLXN_11));
    AND6_MXILINX_fanouts XLXI_7 (.I0(XLXN_11),
                                 .I1(XLXN_11),
                                 .I2(XLXN_11),
                                 .I3(XLXN_11),
                                 .I4(XLXN_11),
                                 .I5(XLXN_11),
                                 .O(XLXN_212));
    // synthesis attribute HU_SET of XLXI_7 is "XLXI_7_3"
    AND16_MXILINX_fanouts XLXI_17 (.I0(XLXN_249),
                                   .I1(XLXN_249),
                                   .I2(XLXN_249),
                                   .I3(XLXN_249),
                                   .I4(XLXN_249),
                                   .I5(XLXN_249),
                                   .I6(XLXN_249),
                                   .I7(XLXN_249),
                                   .I8(XLXN_249),
                                   .I9(XLXN_249),
                                   .I10(XLXN_249),
                                   .I11(XLXN_249),
```

```
                                  .I12(XLXN_249),
                                  .I13(XLXN_249),
                                  .I14(XLXN_249),
                                  .I15(XLXN_249),
                                  .O(XLXN_517));
   // synthesis attribute HU_SET of XLXI_17 is "XLXI_17_9"
   AND8_MXILINX_fanouts XLXI_21 (.I0(XLXN_115),
                                 .I1(XLXN_115),
                                 .I2(XLXN_115),
                                 .I3(XLXN_115),
                                 .I4(XLXN_115),
                                 .I5(XLXN_115),
                                 .I6(XLXN_115),
                                 .I7(XLXN_115),
                                 .O(XLXN_118));
   // synthesis attribute HU_SET of XLXI_21 is "XLXI_21_0"
   AND8_MXILINX_fanouts XLXI_22 (.I0(XLXN_112),
                                 .I1(XLXN_112),
                                 .I2(XLXN_112),
                                 .I3(XLXN_112),
                                 .I4(XLXN_112),
                                 .I5(XLXN_112),
                                 .I6(XLXN_112),
                                 .I7(XLXN_112),
                                 .O(XLXN_144));
   // synthesis attribute HU_SET of XLXI_22 is "XLXI_22_1"
   AND9_MXILINX_fanouts XLXI_23 (.I0(XLXN_152),
                                 .I1(XLXN_152),
                                 .I2(XLXN_152),
                                 .I3(XLXN_152),
                                 .I4(XLXN_152),
                                 .I5(XLXN_152),
                                 .I6(XLXN_152),
                                 .I7(XLXN_152),
                                 .I8(XLXN_152),
                                 .O(XLXN_483));
   // synthesis attribute HU_SET of XLXI_23 is "XLXI_23_2"
   AND9_MXILINX_fanouts XLXI_27 (.I0(XLXN_11),
                                 .I1(XLXN_11),
                                 .I2(XLXN_11),
                                 .I3(XLXN_11),
                                 .I4(XLXN_11),
                                 .I5(XLXN_11),
                                 .I6(XLXN_11),
                                 .I7(XLXN_11),
                                 .I8(XLXN_11),
                                 .O(XLXN_484));
   // synthesis attribute HU_SET of XLXI_27 is "XLXI_27_4"
   AND9_MXILINX_fanouts XLXI_28 (.I0(XLXN_258),
                                 .I1(XLXN_258),
                                 .I2(XLXN_258),
                                 .I3(XLXN_258),
                                 .I4(XLXN_258),
                                 .I5(XLXN_258),
                                 .I6(XLXN_258),
```

```
                                .I7(XLXN_258),
                                .I8(XLXN_258),
                                .O(XLXN_486));
  // synthesis attribute HU_SET of XLXI_28 is "XLXI_28_5"
  AND7_MXILINX_fanouts XLXI_29 (.I0(XLXN_212),
                                .I1(XLXN_212),
                                .I2(XLXN_212),
                                .I3(XLXN_212),
                                .I4(XLXN_212),
                                .I5(XLXN_212),
                                .I6(XLXN_212),
                                .O(XLXN_485));
  // synthesis attribute HU_SET of XLXI_29 is "XLXI_29_6"
  AND8_MXILINX_fanouts XLXI_30 (.I0(XLXN_212),
                                .I1(XLXN_212),
                                .I2(XLXN_212),
                                .I3(XLXN_212),
                                .I4(XLXN_212),
                                .I5(XLXN_212),
                                .I6(XLXN_212),
                                .I7(XLXN_212),
                                .O(XLXN_258));
  // synthesis attribute HU_SET of XLXI_30 is "XLXI_30_7"
  AND8_MXILINX_fanouts XLXI_39 (.I0(XLXN_258),
                                .I1(XLXN_258),
                                .I2(XLXN_258),
                                .I3(XLXN_258),
                                .I4(XLXN_258),
                                .I5(XLXN_258),
                                .I6(XLXN_258),
                                .I7(XLXN_258),
                                .O(XLXN_249));
  // synthesis attribute HU_SET of XLXI_39 is "XLXI_39_8"
  AND9_MXILINX_fanouts XLXI_42 (.I0(XLXN_249),
                                .I1(XLXN_249),
                                .I2(XLXN_249),
                                .I3(XLXN_249),
                                .I4(XLXN_249),
                                .I5(XLXN_249),
                                .I6(XLXN_249),
                                .I7(XLXN_249),
                                .I8(XLXN_249),
                                .O(XLXN_487));
  // synthesis attribute HU_SET of XLXI_42 is "XLXI_42_10"
  AND16_MXILINX_fanouts XLXI_60 (.I0(XLXN_517),
                                 .I1(XLXN_517),
                                 .I2(XLXN_517),
                                 .I3(XLXN_517),
                                 .I4(XLXN_517),
                                 .I5(XLXN_517),
                                 .I6(XLXN_517),
                                 .I7(XLXN_517),
                                 .I8(XLXN_517),
                                 .I9(XLXN_517),
                                 .I10(XLXN_517),
```

```
                                    .I11(XLXN_517),
                                    .I12(XLXN_517),
                                    .I13(XLXN_517),
                                    .I14(XLXN_517),
                                    .I15(XLXN_517),
                                    .O(XLXN_521));
   // synthesis attribute HU_SET of XLXI_60 is "XLXI_60_11"
   AND16_MXILINX_fanouts XLXI_62 (.I0(XLXN_503),
                                    .I1(XLXN_503),
                                    .I2(XLXN_503),
                                    .I3(XLXN_503),
                                    .I4(XLXN_503),
                                    .I5(XLXN_503),
                                    .I6(XLXN_503),
                                    .I7(XLXN_503),
                                    .I8(XLXN_503),
                                    .I9(XLXN_503),
                                    .I10(XLXN_503),
                                    .I11(XLXN_503),
                                    .I12(XLXN_503),
                                    .I13(XLXN_503),
                                    .I14(XLXN_503),
                                    .I15(XLXN_503),
                                    .O(XLXN_522));
   // synthesis attribute HU_SET of XLXI_62 is "XLXI_62_16"
   AND16_MXILINX_fanouts XLXI_63 (.I0(XLXN_506),
                                    .I1(XLXN_506),
                                    .I2(XLXN_506),
                                    .I3(XLXN_506),
                                    .I4(XLXN_506),
                                    .I5(XLXN_506),
                                    .I6(XLXN_506),
                                    .I7(XLXN_506),
                                    .I8(XLXN_506),
                                    .I9(XLXN_506),
                                    .I10(XLXN_506),
                                    .I11(XLXN_506),
                                    .I12(XLXN_506),
                                    .I13(XLXN_506),
                                    .I14(XLXN_506),
                                    .I15(XLXN_506),
                                    .O(XLXN_523));
   // synthesis attribute HU_SET of XLXI_63 is "XLXI_63_18"
   AND16_MXILINX_fanouts XLXI_64 (.I0(XLXN_509),
                                    .I1(XLXN_509),
                                    .I2(XLXN_509),
                                    .I3(XLXN_509),
                                    .I4(XLXN_509),
                                    .I5(XLXN_509),
                                    .I6(XLXN_509),
                                    .I7(XLXN_509),
                                    .I8(XLXN_509),
                                    .I9(XLXN_509),
                                    .I10(XLXN_509),
                                    .I11(XLXN_509),
```

```
                                    .I12(XLXN_509),
                                    .I13(XLXN_509),
                                    .I14(XLXN_509),
                                    .I15(XLXN_509),
                                    .O(XLXN_524));
   // synthesis attribute HU_SET of XLXI_64 is "XLXI_64_12"
   AND16_MXILINX_fanouts XLXI_65 (.I0(XLXN_509),
                                    .I1(XLXN_509),
                                    .I2(XLXN_509),
                                    .I3(XLXN_509),
                                    .I4(XLXN_509),
                                    .I5(XLXN_509),
                                    .I6(XLXN_509),
                                    .I7(XLXN_509),
                                    .I8(XLXN_509),
                                    .I9(XLXN_509),
                                    .I10(XLXN_509),
                                    .I11(XLXN_509),
                                    .I12(XLXN_509),
                                    .I13(XLXN_509),
                                    .I14(XLXN_509),
                                    .I15(XLXN_509),
                                    .O(XLXN_525));
   // synthesis attribute HU_SET of XLXI_65 is "XLXI_65_13"
   AND16_MXILINX_fanouts XLXI_66 (.I0(XLXN_509),
                                    .I1(XLXN_509),
                                    .I2(XLXN_509),
                                    .I3(XLXN_509),
                                    .I4(XLXN_509),
                                    .I5(XLXN_509),
                                    .I6(XLXN_509),
                                    .I7(XLXN_509),
                                    .I8(XLXN_509),
                                    .I9(XLXN_509),
                                    .I10(XLXN_509),
                                    .I11(XLXN_509),
                                    .I12(XLXN_509),
                                    .I13(XLXN_509),
                                    .I14(XLXN_509),
                                    .I15(XLXN_509),
                                    .O(XLXN_603));
   // synthesis attribute HU_SET of XLXI_66 is "XLXI_66_14"
   AND12_MXILINX_fanouts XLXI_67 (.I0(XLXN_520),
                                    .I1(XLXN_520),
                                    .I2(XLXN_517),
                                    .I3(XLXN_517),
                                    .I4(XLXN_517),
                                    .I5(XLXN_517),
                                    .I6(XLXN_517),
                                    .I7(XLXN_517),
                                    .I8(XLXN_517),
                                    .I9(XLXN_517),
                                    .I10(XLXN_517),
                                    .I11(XLXN_517),
                                    .O(XLXN_503));
```

```
   // synthesis attribute HU_SET of XLXI_67 is "XLXI_67_15"
   AND12_MXILINX_fanouts XLXI_69 (.I0(XLXN_537),
                                  .I1(XLXN_503),
                                  .I2(XLXN_503),
                                  .I3(XLXN_503),
                                  .I4(XLXN_503),
                                  .I5(XLXN_503),
                                  .I6(XLXN_503),
                                  .I7(XLXN_503),
                                  .I8(XLXN_503),
                                  .I9(XLXN_503),
                                  .I10(XLXN_503),
                                  .I11(XLXN_503),
                                  .O(XLXN_506));
   // synthesis attribute HU_SET of XLXI_69 is "XLXI_69_17"
   AND12_MXILINX_fanouts XLXI_72 (.I0(XLXN_506),
                                  .I1(XLXN_506),
                                  .I2(XLXN_506),
                                  .I3(XLXN_506),
                                  .I4(XLXN_506),
                                  .I5(XLXN_506),
                                  .I6(XLXN_506),
                                  .I7(XLXN_506),
                                  .I8(XLXN_506),
                                  .I9(XLXN_506),
                                  .I10(XLXN_506),
                                  .I11(XLXN_506),
                                  .O(XLXN_509));
   // synthesis attribute HU_SET of XLXI_72 is "XLXI_72_19"
endmodule
```

FILE: bram2.vf

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.
//////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 9.2.02i
//  \   \         Application : sch2verilog
//  /   /         Filename : bram2.vf
// /___/   /\     Timestamp : 11/11/2007 12:03:10
// \   \  /  \
//  \___\/\___\
//
//Command: C:\Xilinx92i\bin\nt\sch2verilog.exe -intstyle ise -family virtex2 -w
"C:/Documents and Settings/HP_Owner/My
Documents/schoolStuff/Thesis/VHDL/ThesisVHDLSims/bram2.sch" bram2.vf
//Design Name: bram2
//Device: virtex2
//Purpose:
//    This verilog netlist is translated from an ECS schematic.It can be
//    synthesized and simulated, but it should not be modified.
//
```

```
`timescale 1ns / 1ps

module M2_1_MXILINX_bram2(D0,
                          D1,
                          S0,
                          O);

    input D0;
    input D1;
    input S0;
   output O;

   wire M0;
   wire M1;

   AND2B1 I_36_7 (.I0(S0),
                  .I1(D0),
                  .O(M0));
   OR2 I_36_8 (.I0(M1),
               .I1(M0),
               .O(O));
   AND2 I_36_9 (.I0(D1),
                .I1(S0),
                .O(M1));
endmodule
`timescale 1ns / 1ps

module bram2(Add,
             CLK,
             D_out);

    input [14:0] Add;
    input CLK;
   output D_out;

   wire [0:0] XLXN_3;
   wire XLXN_6;
   wire XLXN_8;
   wire XLXN_9;
   wire XLXN_11;
   wire [0:0] XLXN_15;
   wire [0:0] XLXN_16;
   wire [0:0] XLXN_17;

   RAMB16_S1 XLXI_3 (.ADDR(Add[13:0]),
                     .CLK(CLK),
                     .DI(XLXN_3[0]),
                     .EN(XLXN_6),
                     .SSR(XLXN_11),
                     .WE(XLXN_11),
                     .DO(XLXN_16[0]));
   defparam XLXI_3.INIT = 1'h0;
   defparam XLXI_3.INIT_00 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_01 =
```

```
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_02 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_03 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;

---------- OMITTED PARTS of ROM initialization FOR BREVITY -----------------

   defparam XLXI_3.INIT_39 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3A =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3B =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3C =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3D =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3E =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.INIT_3F =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_3.SRVAL = 1'h0;
   defparam XLXI_3.WRITE_MODE = "WRITE_FIRST";
   RAMB16_S1 XLXI_4 (.ADDR(Add[13:0]),
                     .CLK(CLK),
                     .DI(XLXN_17[0]),
                     .EN(XLXN_8),
                     .SSR(XLXN_9),
                     .WE(XLXN_9),
                     .DO(XLXN_15[0]));
   defparam XLXI_4.INIT = 1'h0;
   defparam XLXI_4.INIT_00 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.INIT_01 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.INIT_02 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.INIT_03 =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.INIT_04 =

---------- OMITTED PARTS of ROM initialization FOR BREVITY -----------------

         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.INIT_3F =
         256'h0000000000000000000000000000000000000000000000000000000000000000;
   defparam XLXI_4.SRVAL = 1'h0;
   defparam XLXI_4.WRITE_MODE = "WRITE_FIRST";
   M2_1_MXILINX_bram2 XLXI_5 (.D0(XLXN_15[0]),
                              .D1(XLXN_16[0]),
                              .S0(Add[14]),
                              .O(D_out));
   // synthesis attribute HU_SET of XLXI_5 is "XLXI_5_0"
   GND XLXI_6 (.G(XLXN_11));
```

```
   GND XLXI_7 (.G(XLXN_9));
   GND XLXI_8 (.G(XLXN_17[0]));
   GND XLXI_9 (.G(XLXN_3[0]));
   VCC XLXI_10 (.P(XLXN_6));
   VCC XLXI_11 (.P(XLXN_8));
endmodule
```

FILE: ramtester.vf

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.
//////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 9.2.02i
//  \   \         Application : sch2verilog
//  /   /         Filename : ramtester.vf
// /___/   /\     Timestamp : 11/11/2007 12:03:07
// \   \  /  \
//  \___\/\___\
//
//Command: C:\Xilinx92i\bin\nt\sch2verilog.exe -intstyle ise -family virtex2 -w
"C:/Documents and Settings/HP_Owner/My
Documents/schoolStuff/Thesis/VHDL/ThesisVHDLSims/ramtester.sch" ramtester.vf
//Design Name: ramtester
//Device: virtex2
//Purpose:
//    This verilog netlist is translated from an ECS schematic.It can be
//    synthesized and simulated, but it should not be modified.
//
`timescale 1ns / 1ps

module ramtester(XLXN_1,
                 XLXN_2,
                 XLXN_3,
                 XLXN_4,
                 XLXN_5,
                 XLXN_20,
                 XLXN_21,
                 XLXN_22,
                 XLXN_25,
                 XLXN_26,
                 XLXN_23);

    input XLXN_1;
    input XLXN_2;
    input XLXN_3;
    input XLXN_4;
    input XLXN_5;
    input XLXN_20;
    input XLXN_21;
    input XLXN_22;
    input XLXN_25;
    input XLXN_26;
```

```
   output XLXN_23;


   RAM128X1S XLXI_4 (.A0(XLXN_20),
                     .A1(XLXN_1),
                     .A2(XLXN_2),
                     .A3(XLXN_3),
                     .A4(XLXN_4),
                     .A5(XLXN_5),
                     .A6(XLXN_22),
                     .D(XLXN_26),
                     .WCLK(XLXN_21),
                     .WE(XLXN_25),
                     .O(XLXN_23));
   defparam XLXI_4.INIT = 128'h00000000000000000000000000000000;
endmodule
```

2. Synthesis Reports

The synthesis reports were generated from the VHDL files above. They were generated for the Xilinx Virtex-II XC26000 with package ff1517 and with a speed grade of -4. These reports were used to gather timing and hardware utilization parameters. The key parts that were analyzed were the number of LUTs and Slices and the worst case signal propagation path. The delay due the IOBs was subtracted from the total delay at the end of each synthesis report so that multiple components can be cascaded inside the FPGA. Since the VHDL files were modified without changing the names, often the name of the synthesis report does not reflect the actual size of the device. For example, adder_64.syr, shown below is the synthesis report for a 129-bit RCA.

Parts of the reports have been omitted in this appendix for the sake of brevity. The first synthesis report (for adder_64.syr) shows almost everything that is included in a synthesis report. The following synthesis reports show only information that is pertinent to this thesis.

FILE: adder_64.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.
--> Parameter TMPDIR set to __projnav
CPU : 0.00 / 0.51 s | Elapsed : 0.00 / 0.00 s

--> Parameter xsthdpdir set to ./xst
```

```
CPU : 0.00 / 0.51 s | Elapsed : 0.00 / 0.00 s

--> Reading design: adder_64.prj

TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) HDL Synthesis
  5) Advanced HDL Synthesis
     5.1) HDL Synthesis Report
  6) Low Level Synthesis
  7) Final Report
     7.1) Device utilization summary
     7.2) TIMING REPORT


=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                    : adder_64.prj
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO
Verilog Include Directory          :

---- Target Parameters
Output File Name                   : adder_64
Output Format                      : NGC
Target Device                      : xc2v6000-4-ff1517

---- Source Options
Top Module Name                    : adder_64
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Auto
FSM Style                          : lut
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
ROM Style                          : Auto
Mux Extraction                     : YES
Mux Style                          : Auto
Decoder Extraction                 : YES
Priority Encoder Extraction        : YES
Shift Register Extraction          : YES
Logical Shifter Extraction         : YES
XOR Collapsing                     : YES
Resource Sharing                   : YES
Multiplier Style                   : auto
Automatic Register Balancing       : No

---- Target Options
Add IO Buffers                     : YES
Global Maximum Fanout              : 500
Add Generic Clock Buffer(BUFG)     : 16
```

```
Register Duplication                : YES
Equivalent register Removal         : YES
Slice Packing                       : YES
Pack IO Registers into IOBs         : auto

---- General Options
Optimization Goal                   : Speed
Optimization Effort                 : 1
Keep Hierarchy                      : NO
Global Optimization                 : AllClockNets
RTL Output                          : Yes
Write Timing Constraints            : NO
Hierarchy Separator                 : _
Bus Delimiter                       : <>
Case Specifier                      : maintain
Slice Utilization Ratio             : 100
Slice Utilization Ratio Delta       : 5

---- Other Options
lso                                 : adder_64.lso
Read Cores                          : YES
cross_clock_analysis                : NO
verilog2001                         : YES
Optimize Instantiated Primitives    : NO
tristate2logic                      : No

=========================================================================


=========================================================================
*                           HDL Compilation                             *
=========================================================================
Compiling vhdl file H:/Thesis/VHDL/ThesisVHDLSims/Adder_64.vhd in Library work.
Architecture behavioral of Entity adder_64 is up to date.

=========================================================================
*                            HDL Analysis                               *
=========================================================================
Analyzing Entity <adder_64> (Architecture <behavioral>).
Entity <adder_64> analyzed. Unit <adder_64> generated.


=========================================================================
*                           HDL Synthesis                               *
=========================================================================

Synthesizing Unit <adder_64>.
    Related source file is H:/Thesis/VHDL/ThesisVHDLSims/Adder_64.vhd.
    Found 129-bit adder for signal <sum>.
    Summary:
      inferred   1 Adder/Subtracter(s).
Unit <adder_64> synthesized.


=========================================================================
```

```
*                      Advanced HDL Synthesis                       *
=========================================================================

Advanced RAM inference ...
Advanced multiplier inference ...
Advanced Registered AddSub inference ...
Dynamic shift register inference ...

=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors               : 1
 129-bit adder                     : 1

=========================================================================

=========================================================================
*                        Low Level Synthesis                            *
=========================================================================

Optimizing unit <adder_64> ...
Loading device for application Xst from file '2v6000.nph' in environment
C:/Xilinx.

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block adder_64, actual ratio is 0.

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : adder_64.ngr
Top Level Output File Name         : adder_64
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 387

Macro Statistics :
# Adders/Subtractors               : 1
#      129-bit adder               : 1

Cell Usage :
# BELS                             : 386
#      GND                         : 1
#      LUT2                        : 129
#      MUXCY                       : 128
#      XORCY                       : 128
# IO Buffers                       : 387
#      IBUF                        : 258
#      OBUF                        : 129
```

```
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of Slices:                       65  out of  33792     0%
 Number of 4 input LUTs:                129  out of  67584     0%
 Number of bonded IOBs:                 387  out of   1104    35%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
No clock signals found in this design

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 14.963ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

-------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               14.963ns (Levels of Logic = 132)
  Source:            a<0> (PAD)
  Destination:       sum<128> (PAD)

  Data Path: a<0> to sum<128>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   0.825   0.517  a_0_IBUF (a_0_IBUF)
     LUT2:I0->O            2   0.439   0.000  adder_64_sum<0>lut (sum_0_OBUF)
     MUXCY:S->O            1   0.298   0.000  adder_64_sum<0>cy
(adder_64_sum<0>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<1>cy
(adder_64_sum<1>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<2>cy
(adder_64_sum<2>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<3>cy
```

```
(adder_64_sum<3>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<4>cy
------------ PARTS OMITTED FOR BREVITY ----------------------

(adder_64_sum<113>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<118>cy
(adder_64_sum<118>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<119>cy
(adder_64_sum<119>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<120>cy
(adder_64_sum<120>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<121>cy
(adder_64_sum<121>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<122>cy
(adder_64_sum<122>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<123>cy
(adder_64_sum<123>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<124>cy
(adder_64_sum<124>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<125>cy
(adder_64_sum<125>_cyo)
     MUXCY:CI->O           1   0.053   0.000  adder_64_sum<126>cy
(adder_64_sum<126>_cyo)
     MUXCY:CI->O           0   0.053   0.000  adder_64_sum<127>cy
(adder_64_sum<127>_cyo)
     XORCY:CI->O           1   1.274   0.517  adder_64_sum<128>_xor
(sum_128_OBUF)
     OBUF:I->O                 4.361          sum_128_OBUF (sum<128>)
    ----------------------------------------
    Total                     14.963ns (13.928ns logic, 1.035ns route)
                                       (93.1% logic, 6.9% route)

=========================================================================
CPU : 18.95 / 19.98 s | Elapsed : 19.00 / 20.00 s

-->

Total memory usage is 144088 kilobytes
```

FILE: fanouts.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

------------ PARTS OMITTED FOR BREVITY ----------------------

Input File Name                    : fanouts.prj

------------ PARTS OMITTED FOR BREVITY ----------------------

Cell Usage :
# BELS                             : 125
```

```
#      AND2                          : 5
#      AND3                          : 9
#      AND4                          : 57
#      GND                           : 19
#      MUXCY                         : 7
#      MUXCY_L                       : 21
#      VCC                           : 7
# IO Buffers                         : 16
#      IBUF                          : 3
#      OBUF                          : 13
# Others                             : 68
#      FMAP                          : 68
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of Slices:                      14  out of  33792     0%
 Number of bonded IOBs:                 16  out of   1104     1%


=========================================================================
TIMING REPORT

------------ PARTS OMITTED FOR BREVITY ---------------------

  Data Path: XLXN_115 to XLXN_524
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             10   0.825   0.885  XLXN_115_IBUF (XLXN_115_IBUF)
     AND2:I1->O            11   0.439   0.909  XLXI_3 (XLXN_112)
     AND3:I2->O            13   0.439   0.955  XLXI_4 (XLXN_152)
     AND4:I3->O            15   0.439   0.989  XLXI_5 (XLXN_11)
     begin scope: 'XLXI_7'
     AND3:I2->O             1   0.439   0.517  I_36_69 (I35)
     AND4:I3->O            15   0.439   0.989  I_36_85 (O)
     end scope: 'XLXI_7'
     begin scope: 'XLXI_30'
     AND4:I3->O             1   0.439   0.517  I_36_127 (S1)
     AND2:I1->O            17   0.439   1.012  I_36_142 (O)
     end scope: 'XLXI_30'
     begin scope: 'XLXI_39'
     AND4:I3->O             1   0.439   0.517  I_36_127 (S1)
     AND2:I1->O            25   0.439   1.069  I_36_142 (O)
     end scope: 'XLXI_39'
     begin scope: 'XLXI_17'
     AND4:I3->O             1   0.439   0.000  I_36_110 (S0)
     MUXCY_L:S->LO          1   0.298   0.000  I_36_2 (C0)
     MUXCY_L:CI->LO         1   0.053   0.000  I_36_129 (C1)
     MUXCY_L:CI->LO         1   0.053   0.000  I_36_147 (C2)
     MUXCY:CI->O           26   0.942   1.072  I_36_165 (O)
     end scope: 'XLXI_17'
```

```
     begin scope: 'XLXI_67'
     AND4:I1->O            1   0.439   0.517  I_36_151 (S2)
     AND3:I2->O           27   0.439   1.075  I_36_177 (O)
     end scope: 'XLXI_67'
     begin scope: 'XLXI_69'
     AND4:I1->O            1   0.439   0.517  I_36_151 (S2)
     AND3:I2->O           28   0.439   1.077  I_36_177 (O)
     end scope: 'XLXI_69'
     begin scope: 'XLXI_72'
     AND4:I1->O            1   0.439   0.517  I_36_151 (S2)
     AND3:I2->O           48   0.439   1.129  I_36_177 (O)
     end scope: 'XLXI_72'
     begin scope: 'XLXI_64'
     AND4:I3->O            1   0.439   0.000  I_36_110 (S0)
     MUXCY_L:S->LO         1   0.298   0.000  I_36_2 (C0)
     MUXCY_L:CI->LO        1   0.053   0.000  I_36_129 (C1)
     MUXCY_L:CI->LO        1   0.053   0.000  I_36_147 (C2)
     MUXCY:CI->O           1   0.942   0.517  I_36_165 (O)
     end scope: 'XLXI_64'
     OBUF:I->O                 4.361          XLXN_524_OBUF (XLXN_524)
    ----------------------------------------
    Total                     30.125ns (15.341ns logic, 14.784ns route)
                                       (50.9% logic, 49.1% route)

=========================================================================
CPU : 6.50 / 7.51 s | Elapsed : 7.00 / 8.00 s
```

FILE: BRAM2.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

------------ PARTS OMITTED FOR BREVITY ----------------------

Input File Name                    : bram2.prj

------------ PARTS OMITTED FOR BREVITY ----------------------

=========================================================================
HDL Synthesis Report


------------ PARTS OMITTED FOR BREVITY ----------------------

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : bram2.ngr
Top Level Output File Name         : bram2
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
```

```
Design Statistics
# IOs                              : 17

Cell Usage :
# BELS                             : 9
#      AND2                        : 1
#      AND2b1                      : 1
#      GND                         : 4
#      OR2                         : 1
#      VCC                         : 2
# RAMS                             : 2
#      RAMB16_S1                   : 2
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 16
#      IBUF                        : 15
#      OBUF                        : 1
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of bonded IOBs:                16  out of   1104     1%
 Number of BRAMs:                       2  out of    144     1%
 Number of GCLKs:                       1  out of     16     6%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
CLK                                | BUFGP                  | 2     |
-----------------------------------+------------------------+-------+

------------ PARTS OMITTED FOR BREVITY ----------------------


  Data Path: XLXI_3 to D_out
                             Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     RAMB16_S1:CLK->DO0    1   2.599   0.517  XLXI_3 (XLXN_16)
     begin scope: 'XLXI_5'
     AND2:I0->O            1   0.439   0.517  I_36_9 (M1)
```

```
     OR2:I0->O             1   0.439   0.517  I_36_8 (O)
     end scope: 'XLXI_5'
     OBUF:I->O                 4.361          D_out_OBUF (D_out)
    ----------------------------------------
    Total                      9.391ns (7.838ns logic, 1.552ns route)
                                       (83.5% logic, 16.5% route)

--------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               7.800ns (Levels of Logic = 5)
  Source:            Add<14> (PAD)
  Destination:       D_out (PAD)

  Data Path: Add<14> to D_out
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             2   0.825   0.701  Add_14_IBUF (Add_14_IBUF)
     begin scope: 'XLXI_5'
     AND2b1:I0->O          1   0.439   0.517  I_36_7 (M0)
     OR2:I1->O             1   0.439   0.517  I_36_8 (O)
     end scope: 'XLXI_5'
     OBUF:I->O                 4.361          D_out_OBUF (D_out)
    ----------------------------------------
    Total                      7.800ns (6.064ns logic, 1.736ns route)
                                       (77.7% logic, 22.3% route)

==========================================================================
CPU : 7.44 / 8.47 s | Elapsed : 7.00 / 8.00 s

------------ PARTS OMITTED FOR BREVITY ----------------------
```

FILE: multiplier.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

------------ PARTS OMITTED FOR BREVITY ----------------------

Input File Name                    : multiplier.prj

------------ PARTS OMITTED FOR BREVITY ----------------------

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : multiplier.ngr
Top Level Output File Name         : multiplier
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO
```

```
Design Statistics
# IOs                              : 68

Macro Statistics :
# Multipliers                      : 1
#      17x17-bit multiplier        : 1

Cell Usage :
# BELS                             : 1
#      GND                         : 1
# IO Buffers                       : 68
#      IBUF                        : 34
#      OBUF                        : 34
# MULTs                            : 1
#      MULT18X18                   : 1
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of bonded IOBs:                  68  out of   1104     6%
 Number of MULT18X18s:                    1  out of    144     0%


=========================================================================
TIMING REPORT

------------ PARTS OMITTED FOR BREVITY ----------------------

All values displayed in nanoseconds (ns)

-------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               16.163ns (Levels of Logic = 3)
  Source:            a<0> (PAD)
  Destination:       sum<33> (PAD)

  Data Path: a<0> to sum<33>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   0.825   0.517  a_0_IBUF (a_0_IBUF)
     MULT18X18:A0->P33     1   9.942   0.517  Mmult_sum_inst_mult_0
(sum_33_OBUF)
     OBUF:I->O                 4.361          sum_33_OBUF (sum<33>)
    ----------------------------------------
    Total                     16.163ns (15.128ns logic, 1.035ns route)
                                       (93.6% logic, 6.4% route)

=========================================================================
CPU : 4.75 / 5.80 s | Elapsed : 5.00 / 6.00 s

------------ PARTS OMITTED FOR BREVITY ----------------------
```

FILE: mux128to1.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

------------ PARTS OMITTED FOR BREVITY ----------------------

Input File Name                    : mux128to1.prj

------------ PARTS OMITTED FOR BREVITY ----------------------

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : mux128to1.ngr
Top Level Output File Name         : mux128to1
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 137

Cell Usage :
# BELS                             : 349
#      AND2                        : 64
#      AND2b1                      : 64
#      AND3                        : 14
#      AND3b1                      : 14
#      LUT1                        : 66
#      MUXF5                       : 1
#      MUXF5_L                     : 32
#      MUXF6                       : 16
#      OR2                         : 78
# IO Buffers                       : 137
#      IBUF                        : 136
#      OBUF                        : 1
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of Slices:                       33  out of  33792     0%
 Number of 4 input LUTs:                 66  out of  67584     0%
 Number of bonded IOBs:                 137  out of   1104    12%


=========================================================================
```

```
TIMING REPORT

------------ PARTS OMITTED FOR BREVITY ---------------------

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

--------------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               17.386ns (Levels of Logic = 21)
  Source:            Sel<0> (PAD)
  Destination:       XLXN_20 (PAD)

  Data Path: Sel<0> to XLXN_20
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           128   0.825   1.316  Sel_0_IBUF (Sel_0_IBUF)
     begin scope: 'XLXI_3_XLXI_2'
     begin scope: 'I_MAB'
     AND2b1:I0->O          1   0.439   0.517  I_36_7 (M0)
     OR2:I1->O             1   0.439   0.517  I_36_8 (O)
     end scope: 'I_MAB'
     LUT1:I0->O            1   0.439   0.000  MAB_rt (MAB_rt)
     MUXF5_L:I1->LO        1   0.436   0.000  I_M8B (M8B)
     MUXF6:I0->O           1   0.447   0.517  I_M8F (MBF)
     begin scope: 'I_O'
     AND3:I0->O            1   0.439   0.517  I_36_30 (M1)
     OR2:I0->O             1   0.439   0.517  I_36_38 (O)
     end scope: 'I_O'
     end scope: 'XLXI_3_XLXI_2'
     begin scope: 'XLXI_3_XLXI_4'
     AND3:I0->O            1   0.439   0.517  I_36_30 (M1)
     OR2:I0->O             1   0.439   0.517  I_36_38 (O)
     end scope: 'XLXI_3_XLXI_4'
     begin scope: 'XLXI_6'
     begin scope: 'I_M01'
     AND3:I0->O            1   0.439   0.517  I_36_30 (M1)
     OR2:I0->O             1   0.439   0.517  I_36_38 (O)
     end scope: 'I_M01'
     LUT1:I0->O            1   0.439   0.000  M01_rt (M01_rt)
     MUXF5:I0->O           1   0.436   0.517  I_O (O)
     end scope: 'XLXI_6'
     OBUF:I->O                 4.361          XLXN_20_OBUF (XLXN_20)
    ----------------------------------------
    Total                     17.386ns (10.895ns logic, 6.491ns route)
                                       (62.7% logic, 37.3% route)

================================================================================
CPU : 7.42 / 8.44 s | Elapsed : 7.00 / 8.00 s

------------ PARTS OMITTED FOR BREVITY ---------------------
```

FILE: ramtester.syr

```
Release 6.3.03i - xst G.38
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

------------ PARTS OMITTED FOR BREVITY ----------------------

Input File Name                    : ramtester.prj

------------ PARTS OMITTED FOR BREVITY ----------------------

=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Top Level Output File Name     : ramtester.ngr
Top Level Output File Name         : ramtester
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : NO

Design Statistics
# IOs                              : 11

Cell Usage :
# RAMS                             : 1
#      RAM128X1S                   : 1
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 10
#      IBUF                        : 9
#      OBUF                        : 1
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v6000ff1517-4

 Number of Slices:                        4  out of  33792     0%
 Number of bonded IOBs:                  10  out of   1104     0%
 Number of GCLKs:                         1  out of     16     6%


=========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
```

```
XLXN_21                            | BUFGP                    | 1     |
-----------------------------------+--------------------------+-------+

Timing Summary:

------------ PARTS OMITTED FOR BREVITY ----------------------

  Data Path: XLXN_26 to XLXI_4
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   0.825   0.517  XLXN_26_IBUF (XLXN_26_IBUF)
     RAM128X1S:D               0.727          XLXI_4
    ----------------------------------------
    Total                      2.069ns (1.552ns logic, 0.517ns route)
                                       (75.0% logic, 25.0% route)

--------------------------------------------------------------------------------
Timing constraint: Default OFFSET OUT AFTER for Clock 'XLXN_21'
Offset:              7.682ns (Levels of Logic = 1)
  Source:            XLXI_4 (RAM)
  Destination:       XLXN_23 (PAD)
  Source Clock:      XLXN_21 rising

  Data Path: XLXI_4 to XLXN_23
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     RAM128X1S:WCLK->O     1   2.804   0.517  XLXI_4 (XLXN_23_OBUF)
     OBUF:I->O                 4.361          XLXN_23_OBUF (XLXN_23)
    ----------------------------------------
    Total                      7.682ns (7.165ns logic, 0.517ns route)
                                       (93.3% logic, 6.7% route)

--------------------------------------------------------------------------------
Timing constraint: Default path analysis
Delay:               8.583ns (Levels of Logic = 3)
  Source:            XLXN_20 (PAD)
  Destination:       XLXN_23 (PAD)

  Data Path: XLXN_20 to XLXN_23
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O            16   0.825   1.000  XLXN_20_IBUF (XLXN_20_IBUF)
     RAM128X1S:A0->O       1   1.879   0.517  XLXI_4 (XLXN_23_OBUF)
     OBUF:I->O                 4.361          XLXN_23_OBUF (XLXN_23)
    ----------------------------------------
    Total                      8.583ns (7.065ns logic, 1.518ns route)
                                       (82.3% logic, 17.7% route)

==========================================================================
CPU : 5.50 / 6.51 s | Elapsed : 6.00 / 7.00 s

------------ PARTS OMITTED FOR BREVITY ----------------------
```

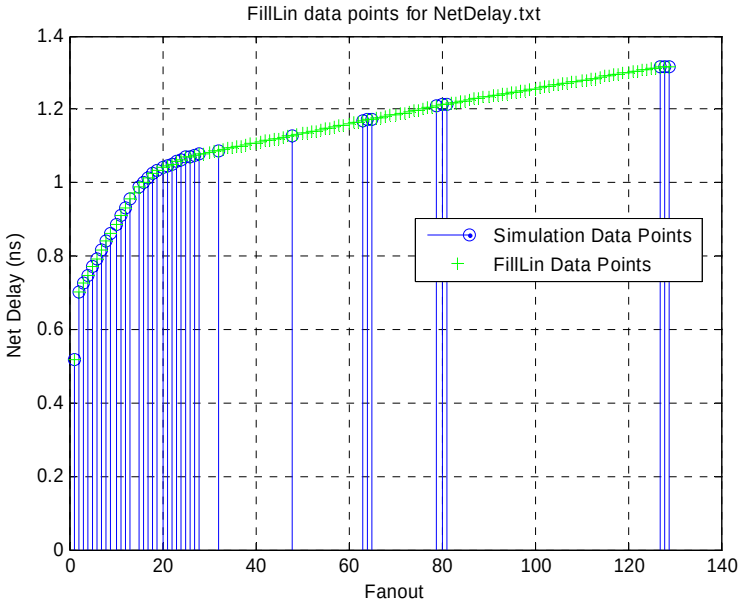
B.2 COLLECTED DATA TEXT FILES

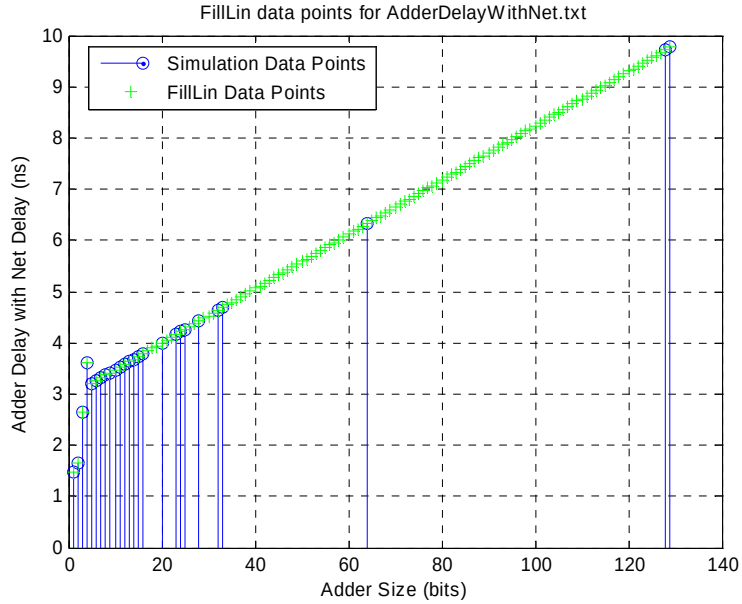
The following data has been collected from synthesis reports and placed into each text file. For each value of *n*, a circuit was synthesized.

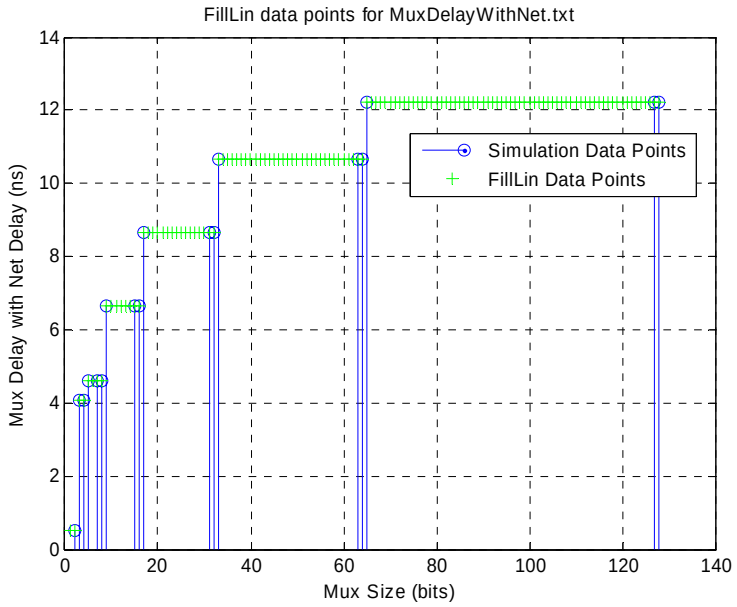
NetDelay.txt	MuxDelayWithNet.txt	AdderDelayWithNet.txt	MultDelayWithNet.txt	MultSlices.txt
n Delay (ns)	n Delay (ns)	n Delay (ns)	n Delay (ns)	n Slices
1 0.517	2 0.517	1 1.474	2 4.766	1 0
2 0.701	3 4.0527	2 1.658	4 5.595	2 0
3 0.725	4 4.0527	3 2.638	6 6.423	3 0
4 0.747	5 4.5917	4 3.617	8 7.251	4 0
5 0.771	7 4.5917	5 3.205	12 8.906	5 0
6 0.794	8 4.5917	6 3.258	16 10.562	6 0
7 0.817	9 6.6657	7 3.311	17 10.977	7 0
8 0.84	15 6.6657	8 3.364	18 16.218	8 0
9 0.863	16 6.6657	9 3.417	19 16.424	12 0
10 0.885	17 8.6657	10 3.47	20 16.43	16 0
11 0.909	31 8.6657	11 3.523	20 16.43	17 0
12 0.931	32 8.6657	12 3.576	21 16.536	18 19
13 0.955	33 10.6617	13 3.629	24 16.854	19 22
15 0.989	63 10.6617	14 3.682	32 17.702	20 24
16 1	64 10.6617	15 3.735	34 17.914	20 24
17 1.012	65 12.1997	16 3.788	35 20.518	21 26
18 1.024	127 12.1997	20 4	36 20.624	24 32
19 1.035	128 12.1997	23 4.159	37 20.73	32 48
20 1.041		24 4.212	51 22.214	34 52
21 1.046		25 4.265	52 22.343	35 89
22 1.052		28 4.424	53 22.449	36 93
23 1.058		32 4.636	54 22.555	37 97
24 1.064		33 4.689	55 22.661	51 146
25 1.069		64 6.332	64 23.615	52 193
26 1.072		128 9.724	68 24.039	53 197
27 1.075		129 9.777	69 26.644	54 203
28 1.077			70 26.75	55 206
32 1.088			85 28.34	64 248
48 1.129			86 28.469	68 266
63 1.168			87 28.575	69 348
64 1.171			102 30.165	70 353
65 1.173			103 30.294	85 445
79 1.209			104 30.4	86 520
80 1.212			119 31.99	87 525
81 1.215			120 32.119	102 633
127 1.316			121 32.225	103 734
128 1.316			128 32.967	104 740
129 1.316			136 33.815	119 863
			137 36.42	120 974
			138 36.526	121 980
				128 1047
				136 1119
				137 1299
				138 1306

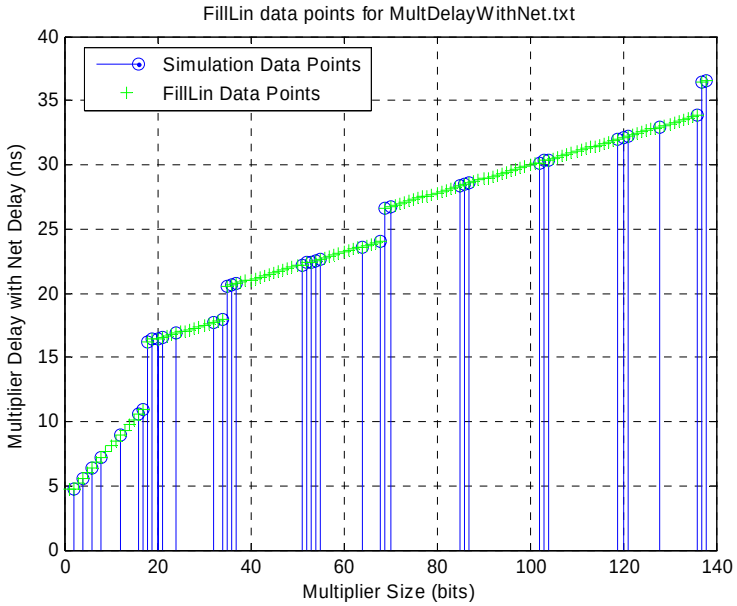
B.3 ESTIMATION OF MISSING DATAPOINTS

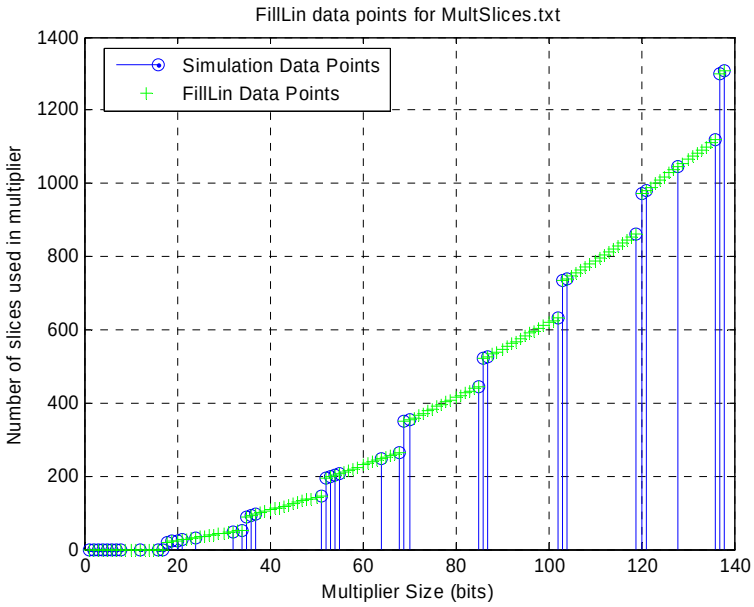
The following plots show how fillLin estimates missing the data points in the five sets of collected data points. The values returned from fillLin are used in HUandDelay to estimate component complexity and delay.






FillLin data points for MuxDelayWithNet.txt
Mux Delay with Net Delay (ns)
Mux Size (bits)
Simulation Data Points
FillLin Data Points
0
2
4
6
8
10
12
14
0
20
40
60
80
100
120
140


FillLin data points for MultDelayWithNet.txt
Multiplier Delay with Net Delay (ns)
Multiplier Size (bits)
Simulation Data Points
FillLin Data Points
0
5
10
15
20
25
30
35
40
0
20
40
60
80
100
120
140


FillLin data points for MultSlices.txt
Simulation Data Points
FillLin Data Points
Number of slices used in multiplier
Multiplier Size (bits)
0
200
400
600
800
1000
1200
1400
0
20
40
60
80
100
120
140

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. COMMONLY USED VARIABLES

C.1 VARIABLE DEFINITIONS

The following is a list of the variables used in this thesis and their descriptions.

Variable	Definition(s)	How determined
ε	Maximum allowable error	Defined by system, here $\varepsilon = 2^{-n-1}$
$\sigma_{\min}$	Minimum segment width	$\sigma_{\min} = 4\sqrt[p]{\frac{\varepsilon}{\left\|f^{(p)}(x^*)\right\|}}$, $p = \begin{cases} 2 & \text{linear} \\ 3 & \text{quadratic} \end{cases}$
c_{2i}, c_{1i}, c_{0i}	Coefficient values for the approximation equation for the i-th segment	Determined by segmentation algorithms.
i	Segment index number	SIE or part of $\boldsymbol{x}$ determines i
k	Number of address lines to the coefficient table of an NFG	$k = \lceil \log_2 s_{\min} \rceil$
n	1. Number of bits in $\boldsymbol{x}$ 2. Bus-width for a given NFG	Defined by NFG requirements
s	number of segments to be used in an NFG	$s = 2^{\lceil \log_2 s_{\min} \rceil} = 2^k$
$s_{\min}$	Minimum number of segments required for an NFG	From segmentation algorithms or by segments.m
SRR	Segment Reduction Ratio	$SRR = \frac{s_{\min}^{non-unif}}{s_{\min}^{unif}}$
t_{prop}	Combinational propagation delay through a logic device	Using models or HUandDelay.m
$x_{\max,i}$	Maximum value of $\boldsymbol{x}$ in segment i	From segmentation algorithms
$x_{\min,i}$	Minimum value of $\boldsymbol{x}$ in segment i	From segmentation algorithms
y	Approximation function, linear or quadratic	Defined by NFG architecture

Table 13 Variable Definitions.

C.2 COMMON VARIABLE VALUES

The following is a list of parameters used throughout this thesis. These values were extracted from empirical evidence and/or product specifications sheets [18]. The values with the more significant digits was utilized for all calculations.

Parameter	Description	From Simulation	From [18]
$t_{MUXCY,S\rightarrow O}$	Propagation delay from the select line of MUXCY to the output.	0.298 ns	Note 2
$t_{MUXCY,I0\rightarrow O}$	Propagation delay from the either input (I0 or I1) of MUXCY to the output.	0.053 ns	0.05 ns
t_{ORCY}	Referred to as t_{SOPSOP} [18], the propagation delay through the fast SOP OR gate, ORCY.	0.439 ns	0.44 ns
$t_{LUT,4}$	Referred to as t_{ILO} [18], Propagation delay through a 4-input LUT	0.439 ns	0.44 ns
$t_{LUT,5}$	Referred to as t_{IF5} [18], Propagation delay through a 5-input LUT	Note 1	0.72 ns

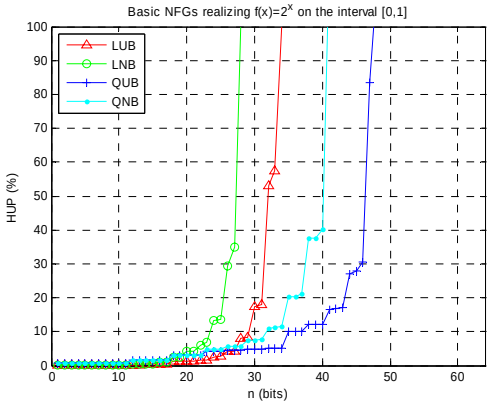
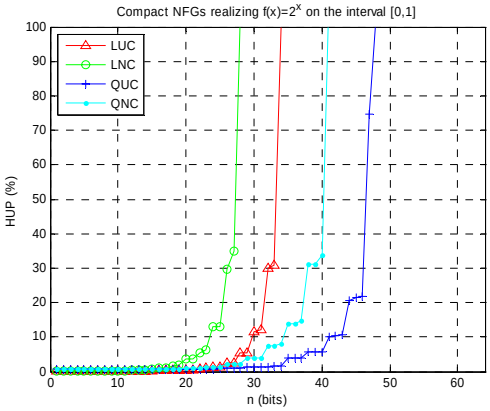
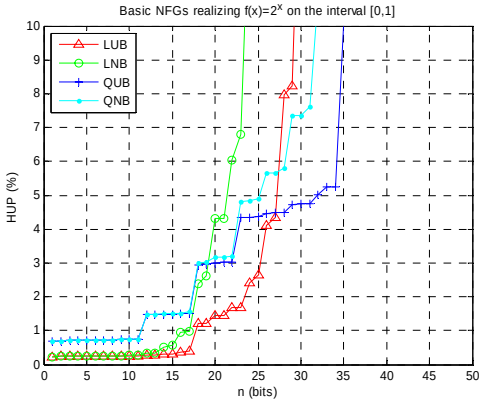
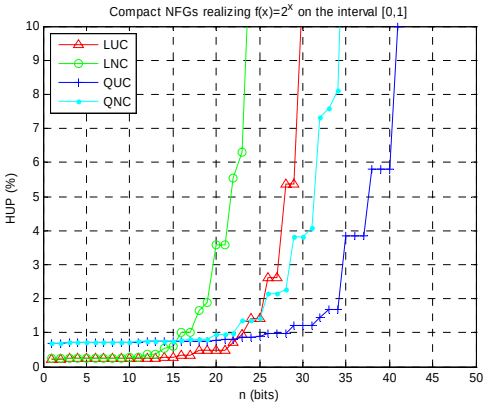
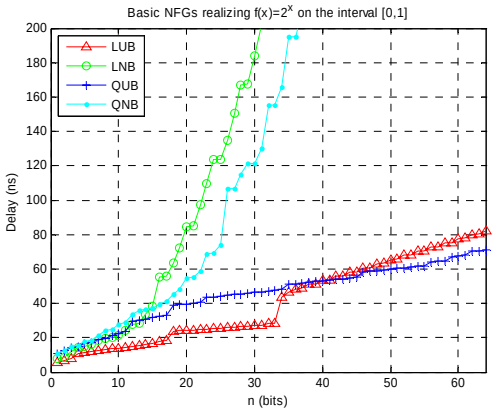
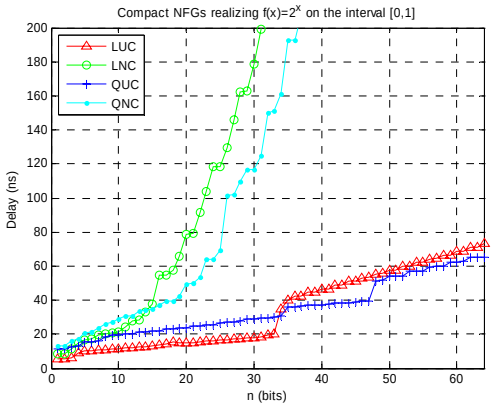
1. No simulation data for this value.
2. Value is not found in reference.

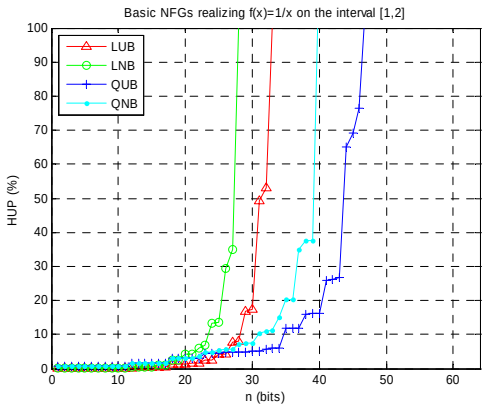
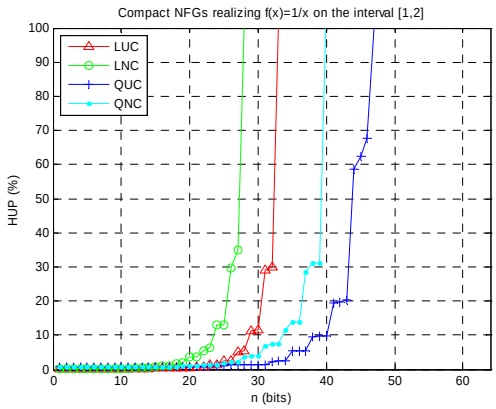
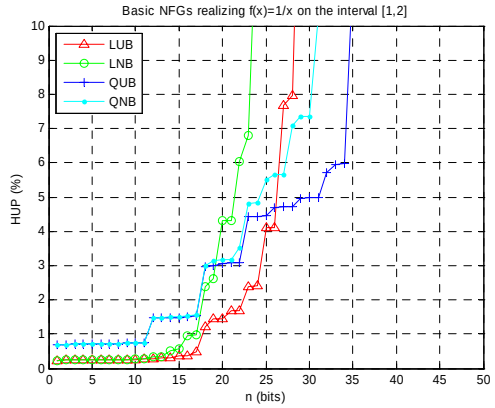
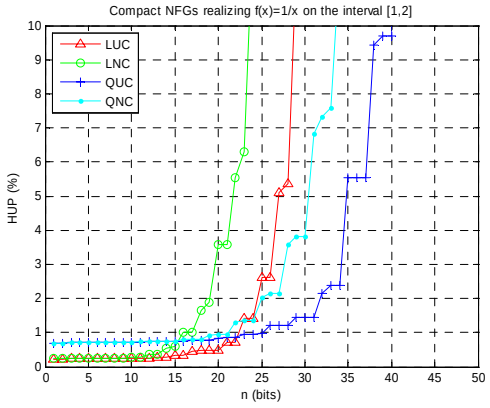
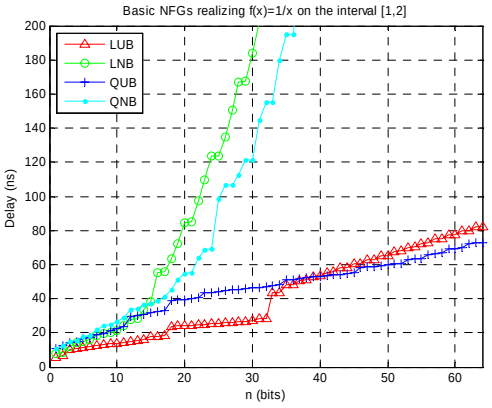
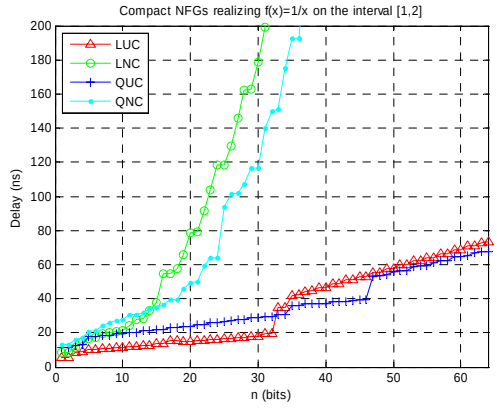
Table 14 Common Variable Values.

APPENDIX D. MODEL DATA

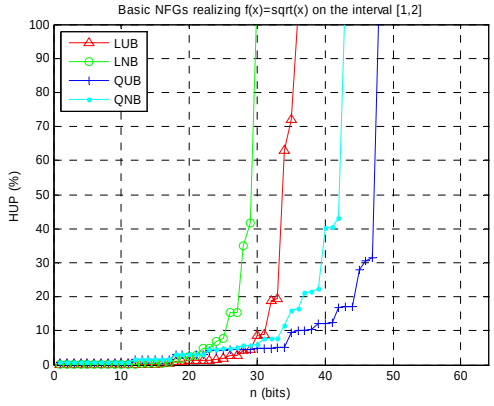
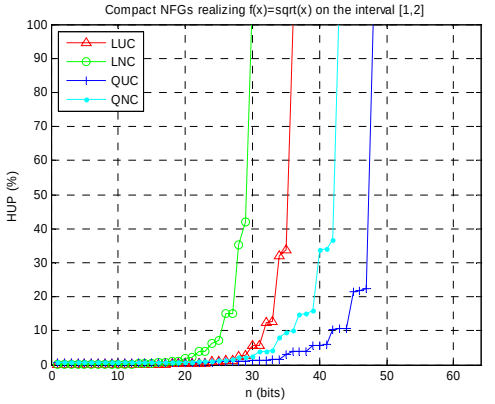
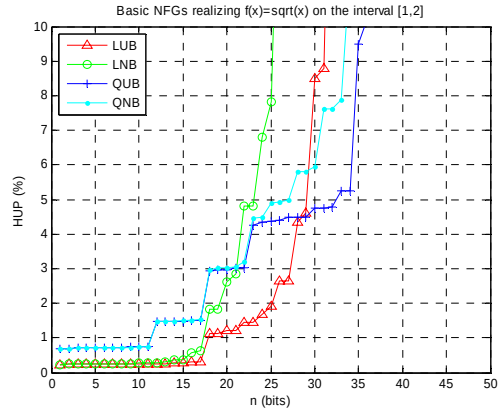
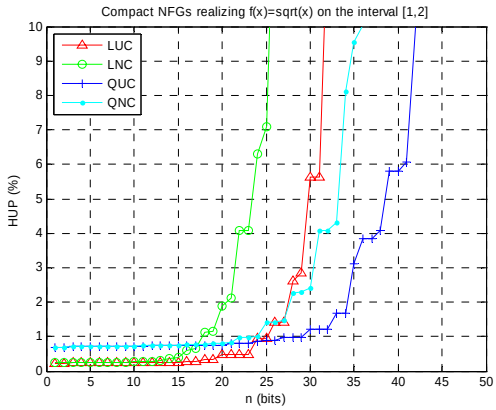
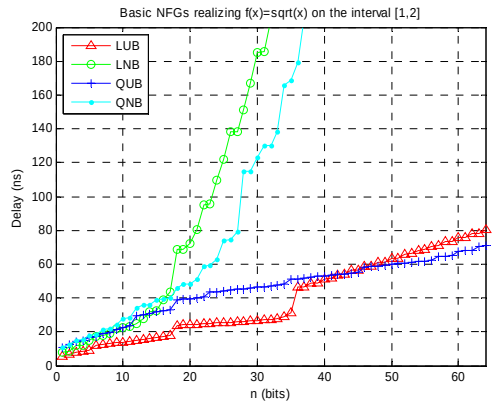
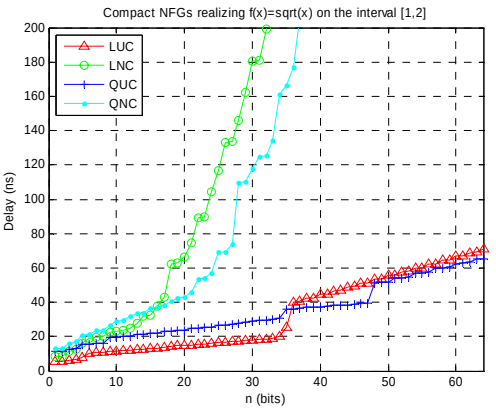
D.1 COMPLEXITY AND DELAY FOR BASIC AND COMPACT NFGS FOR THE FUNCTIONS IN THE FUNCTION SUITE

$f(x) = 2^x$ on [0,1]

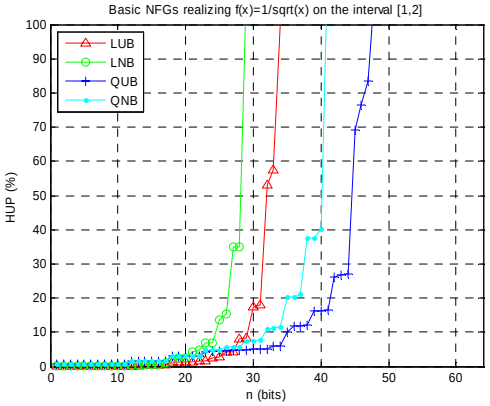
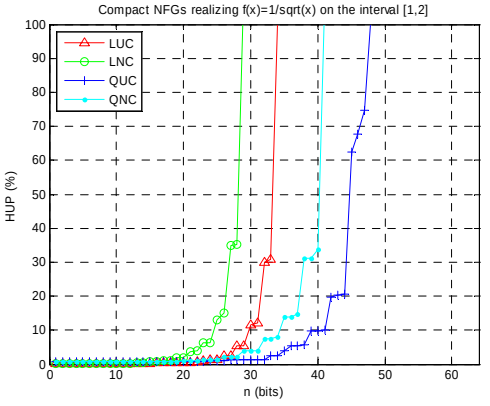
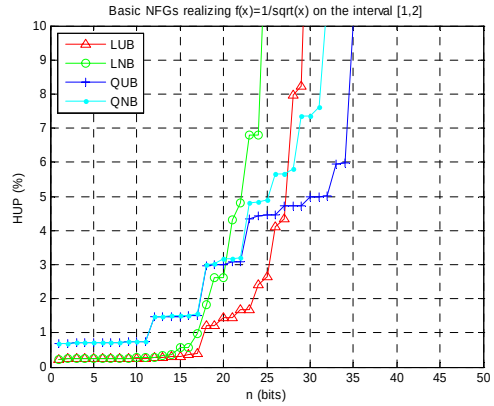
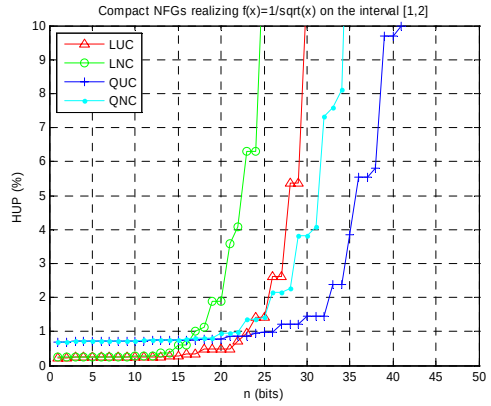
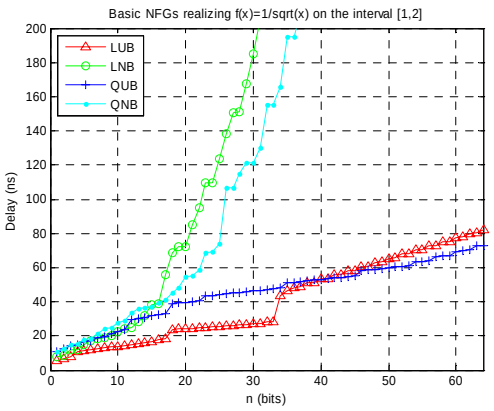
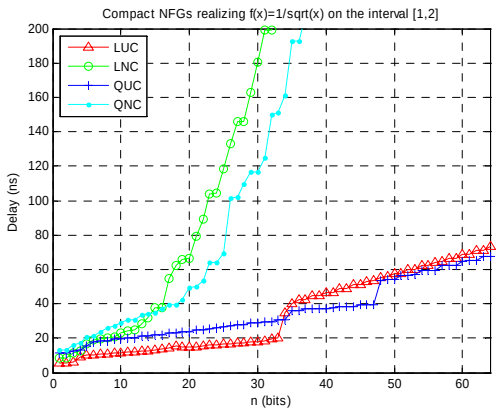
Basic Architectures	Compact Achitectures
	
	
	

$f(x)=1/x$ on [1,2]	
Basic Architectures	**Compact Achitectures**
	
	
	

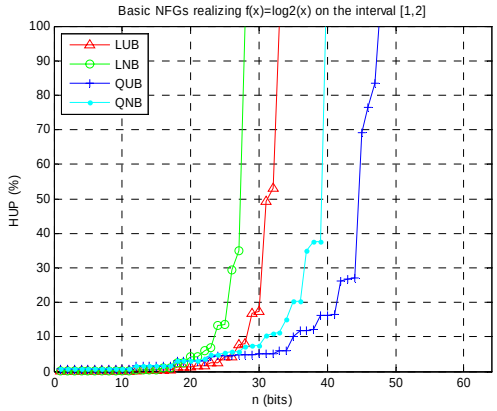
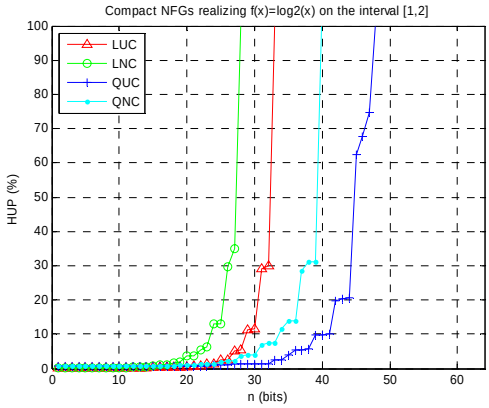
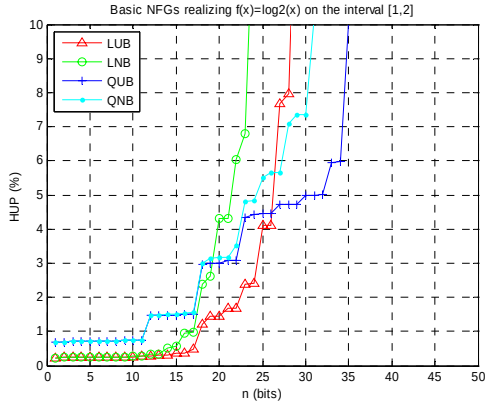
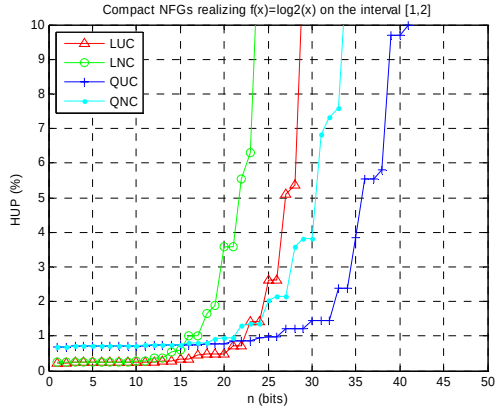
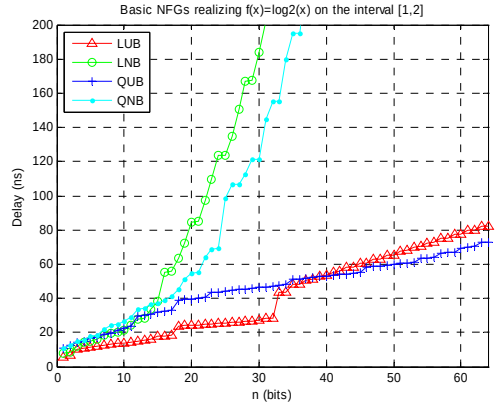
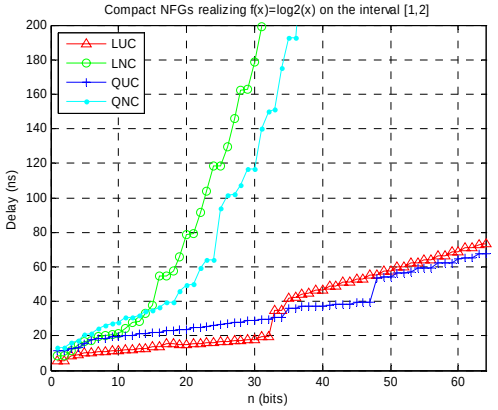
$f(x) = \sqrt{x}$ on [1,2]

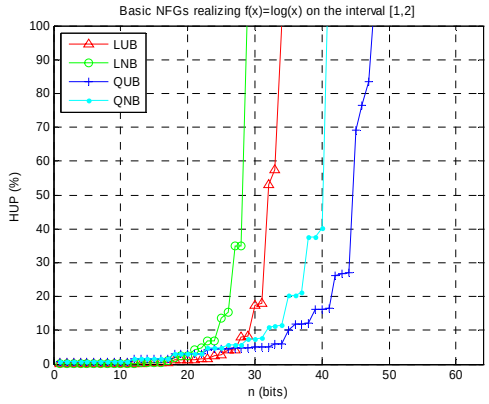
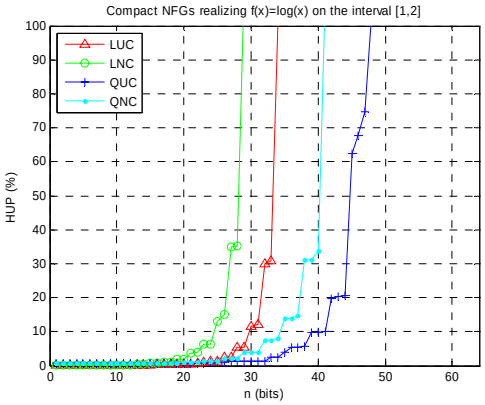
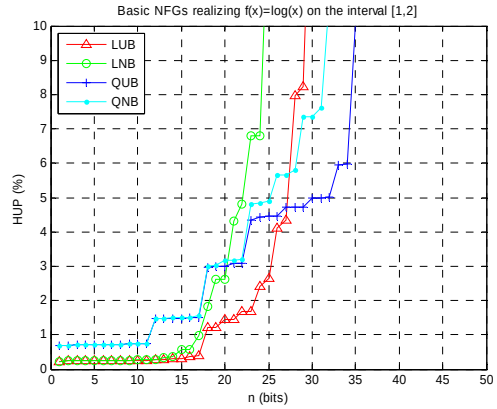
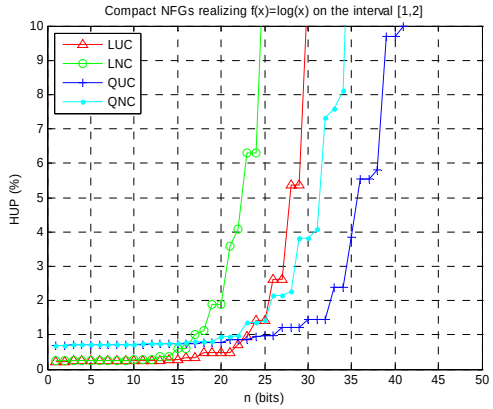
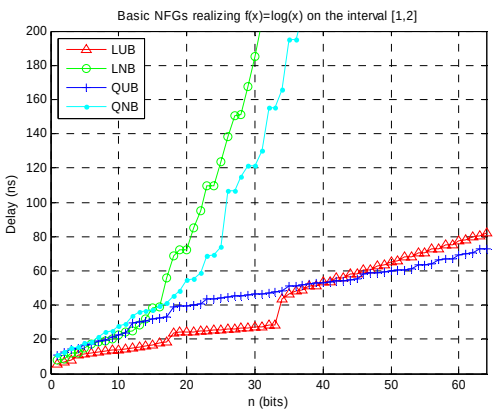
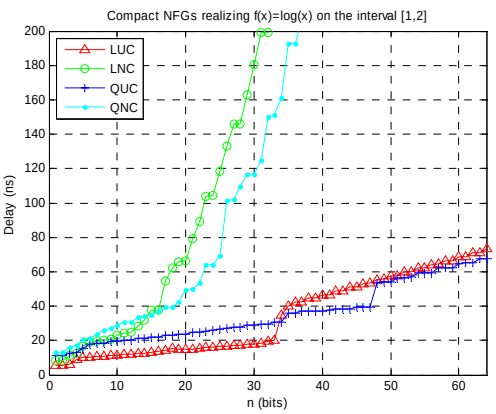
Basic Architectures	Compact Achitectures
	
	
	

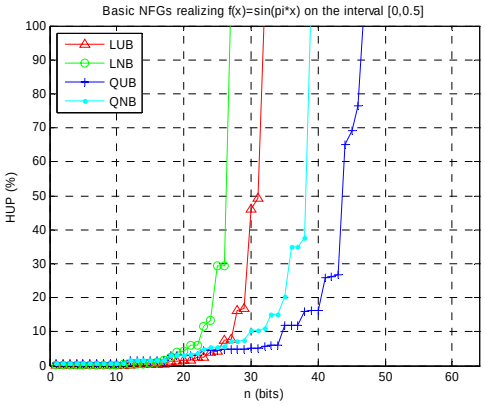
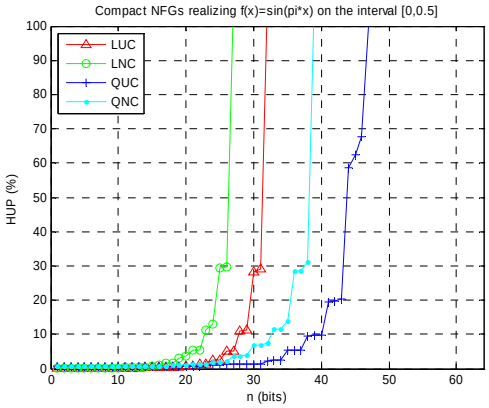
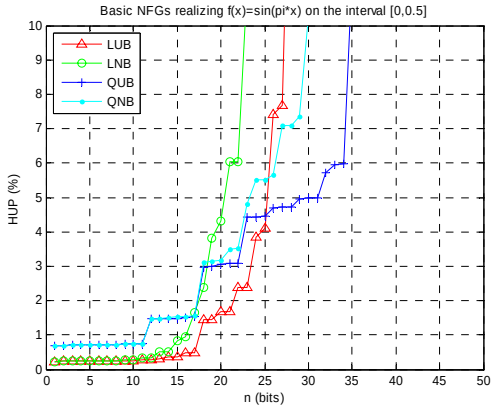
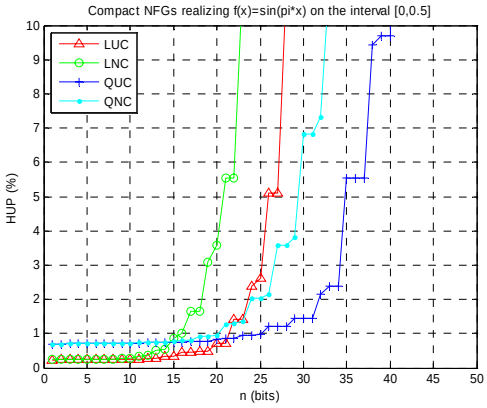
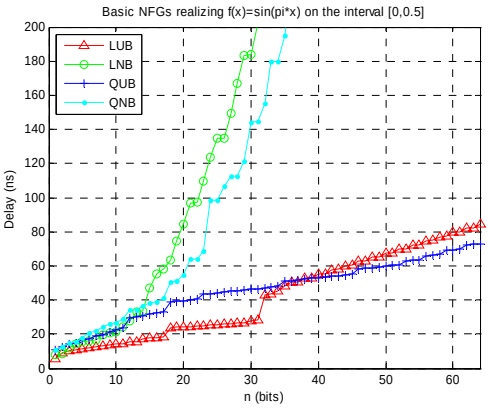
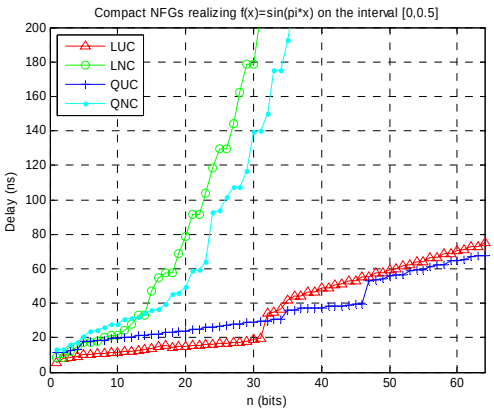
$f(x) = 1/\sqrt{x}$ on [1,2]

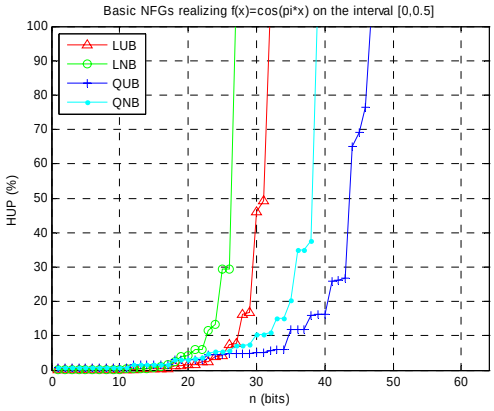
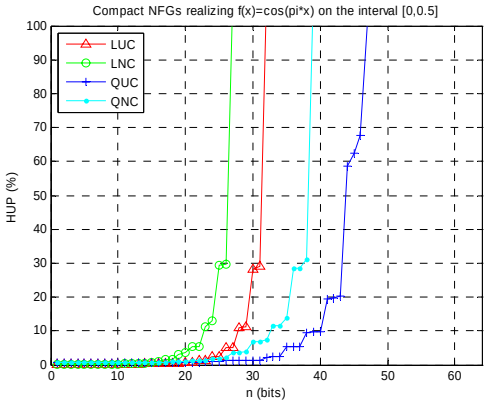
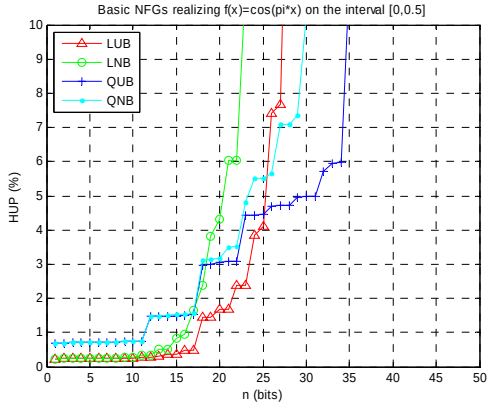
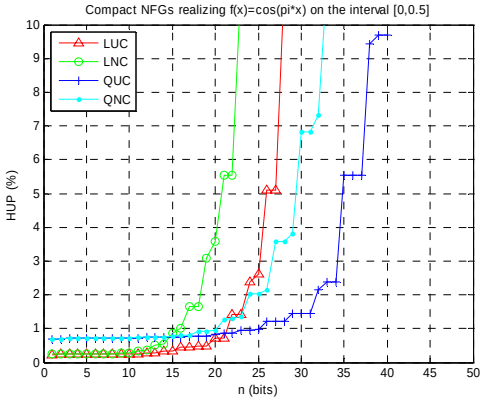
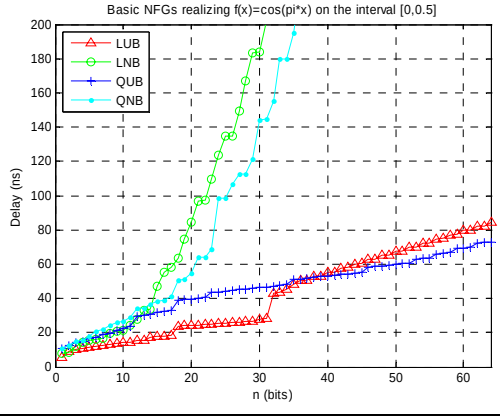
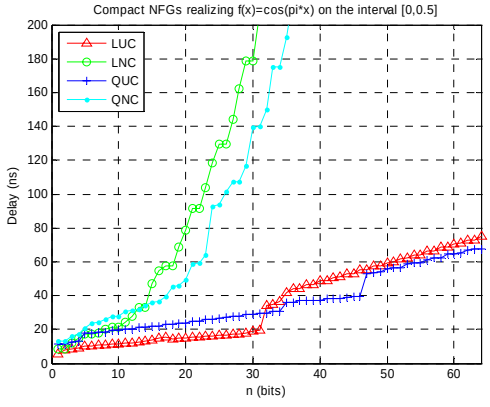
Basic Architectures	Compact Achitectures
	
	
	

$f(x) = \log_2 x$ on [1,2]

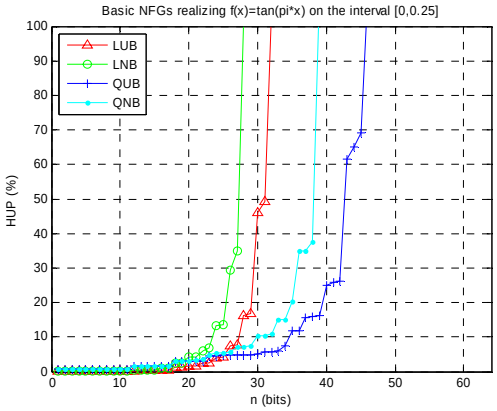
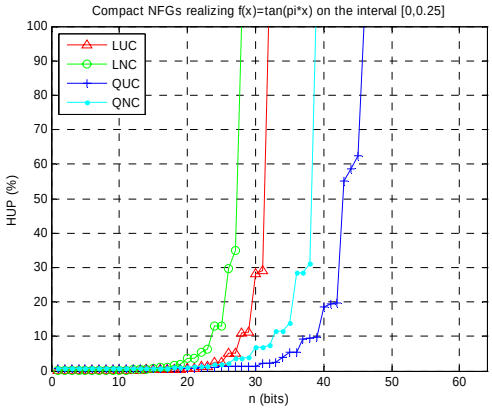
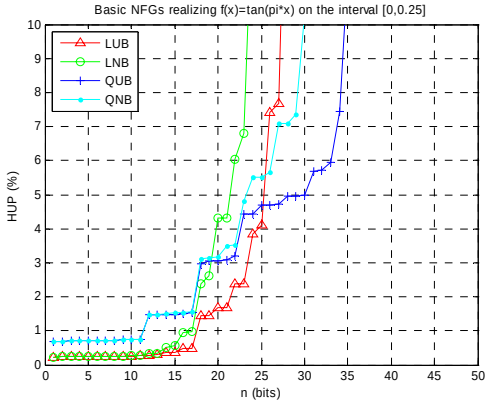
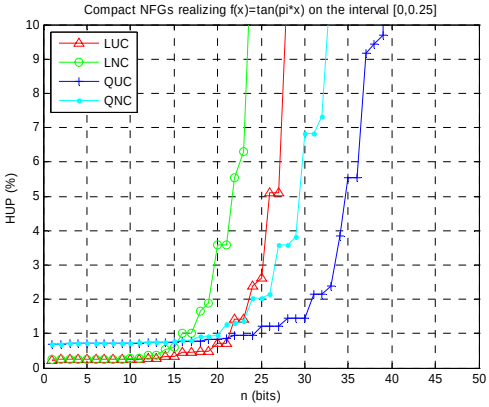
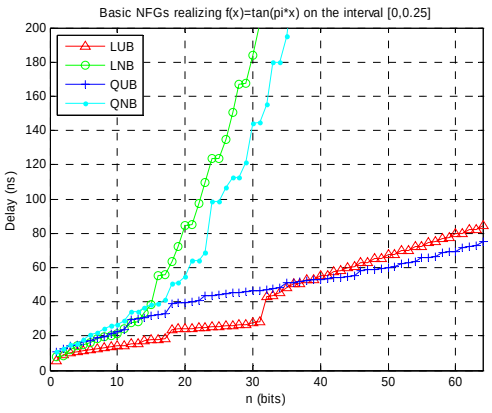
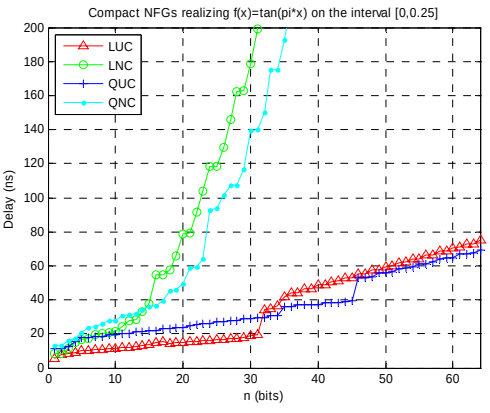
Basic Architectures	Compact Achitectures
	
	
	

$f(x) = \ln(x)$ on [1,2]	
Basic Architectures	**Compact Achitectures**
	
	
	

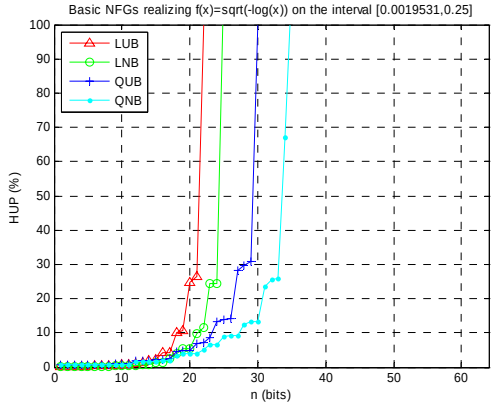
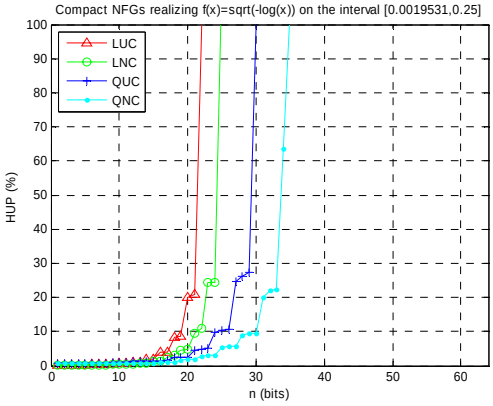
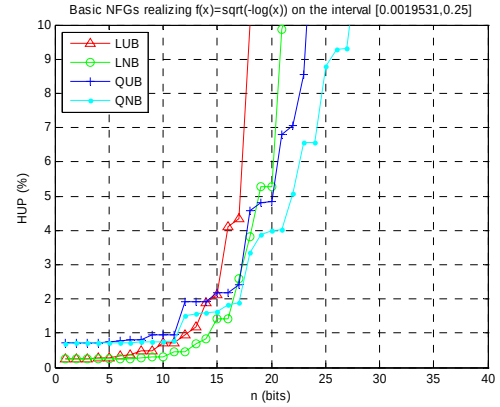
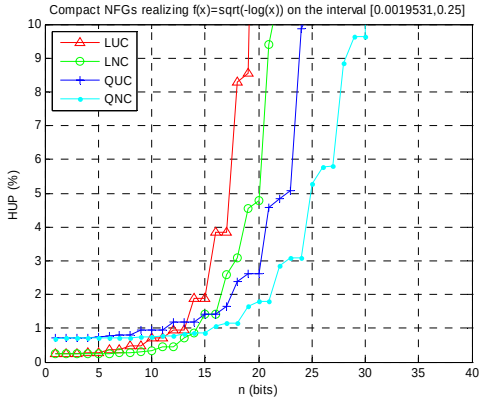
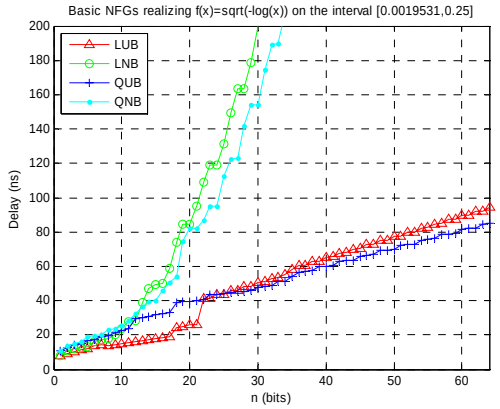
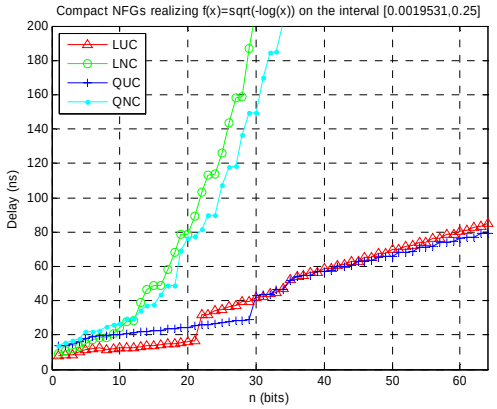
$f(x) = \sin \pi x$ on [0,0.5]	
Basic Architectures	**Compact Achitectures**
	
	
	

$f(x) = \cos \pi x$ on [0,0.5]	
Basic Architectures	**Compact Achitectures**
	
	
	

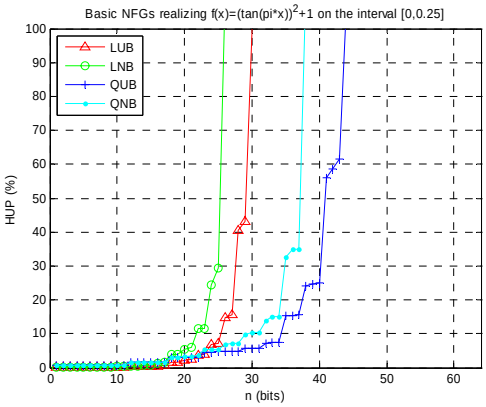
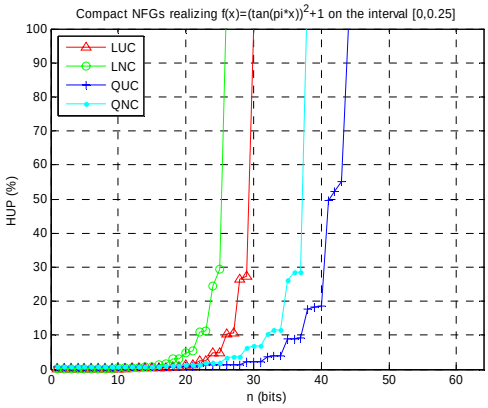
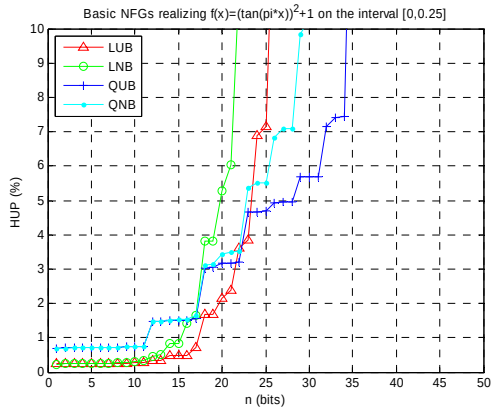
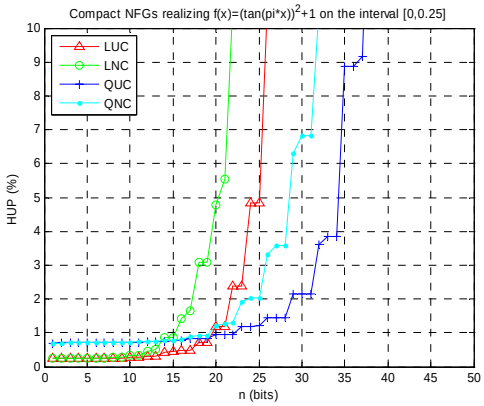
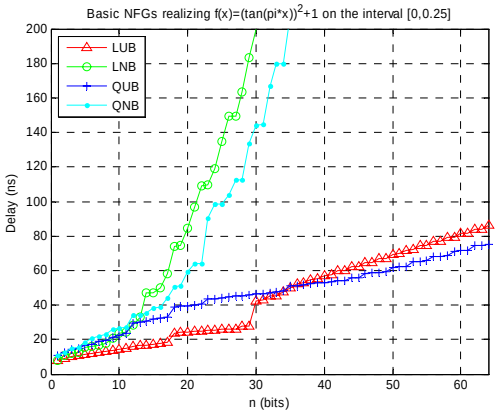
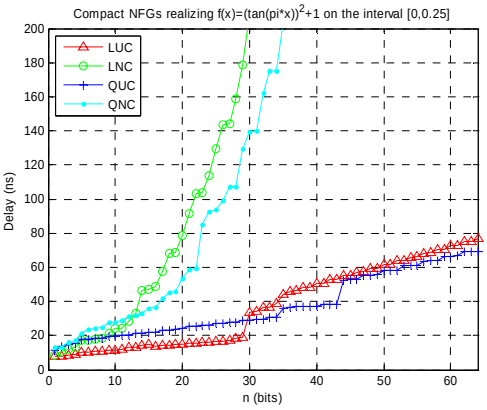
$f(x) = \tan \pi x$ on [0,0.25]

Basic Architectures	Compact Achitectures
	
	
	

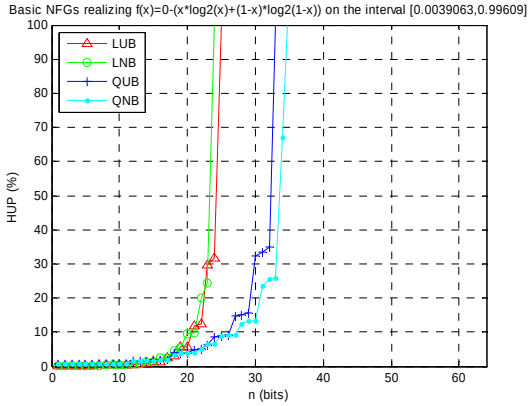
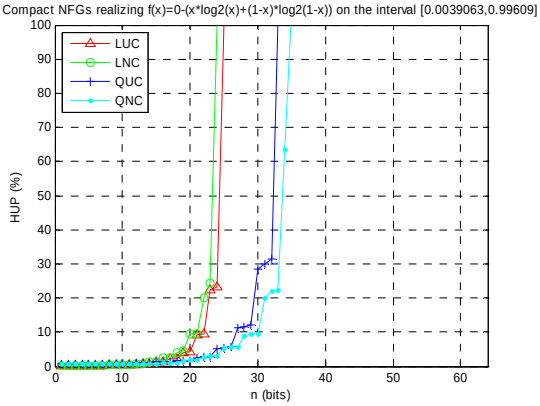
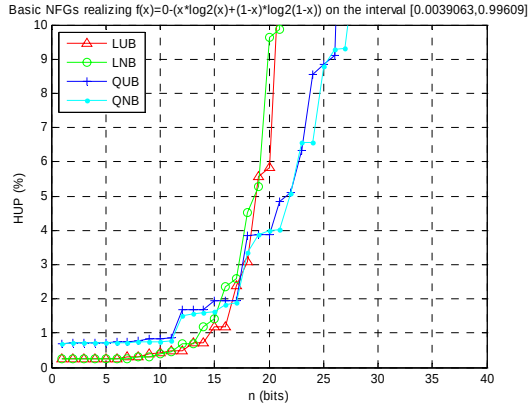
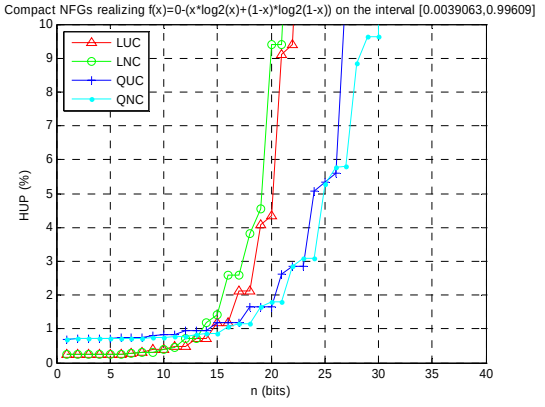
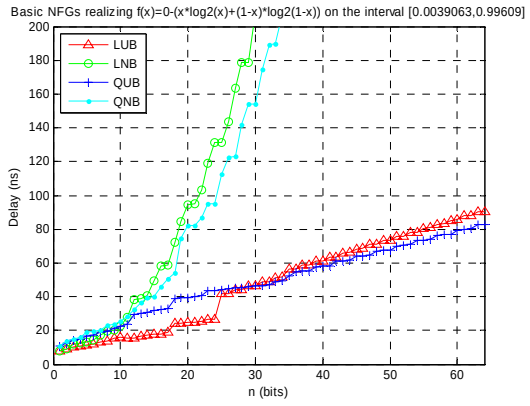
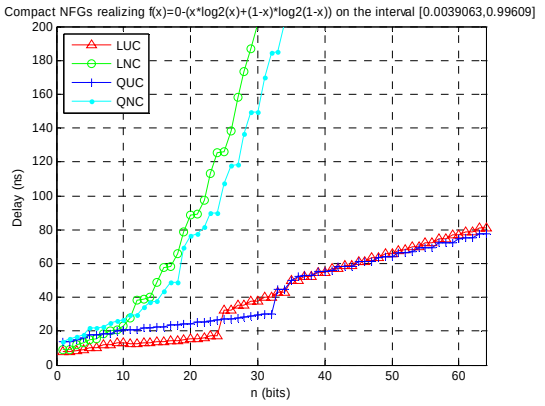
$f(x) = \sqrt{-\ln x}$ on [1/512,1/4]

Basic Architectures	Compact Achitectures
	
	
	

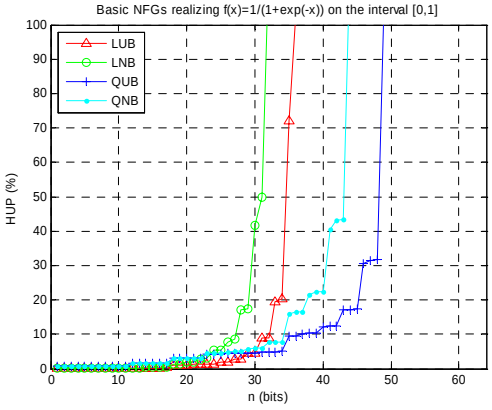
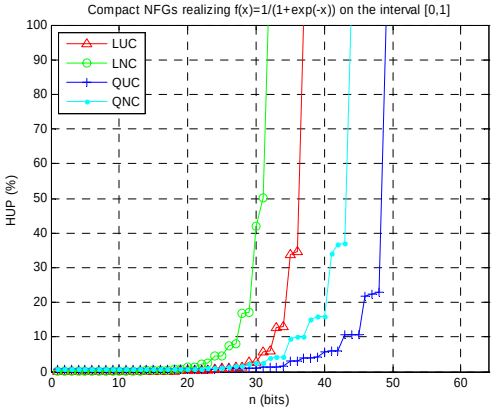
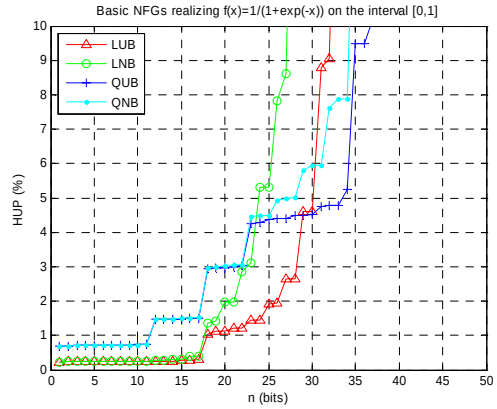
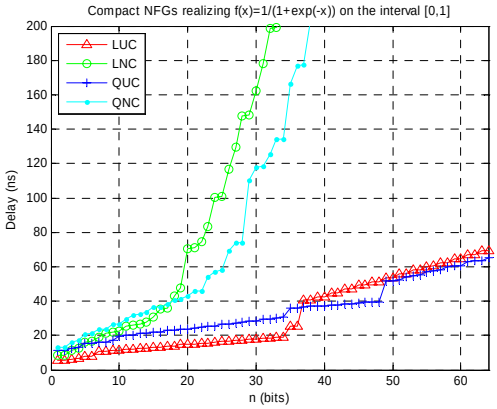
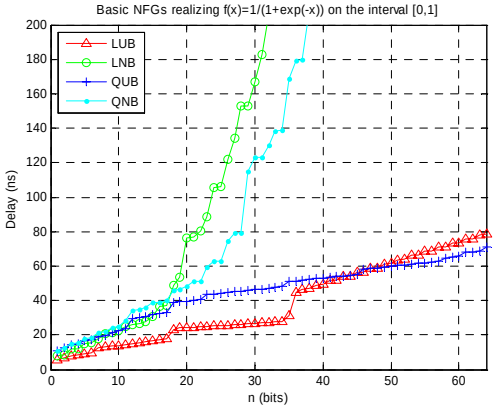
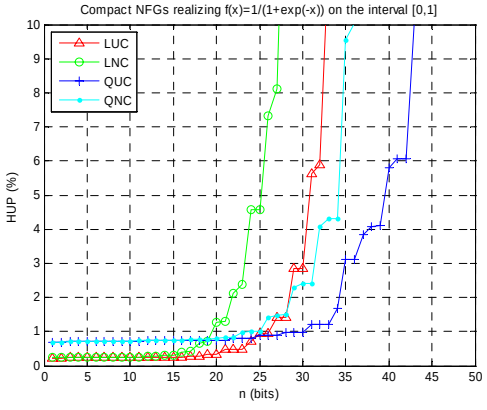
$f(x) = \tan^2 \pi x + 1$ on [0,0.25]

Basic Architectures	Compact Achitectures
	
	
	

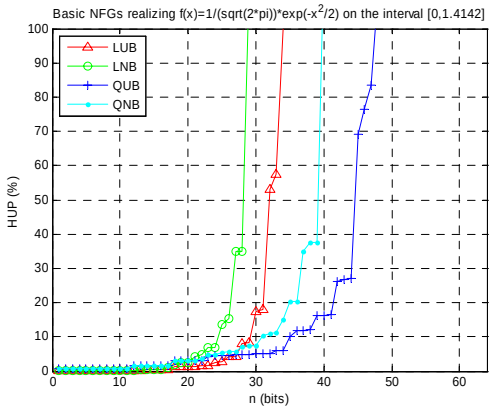
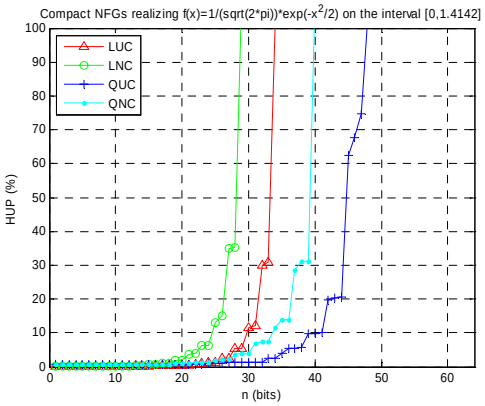
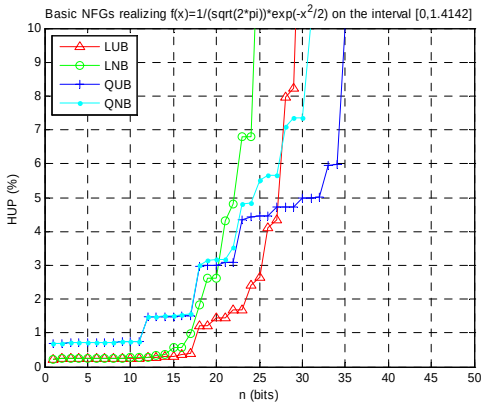
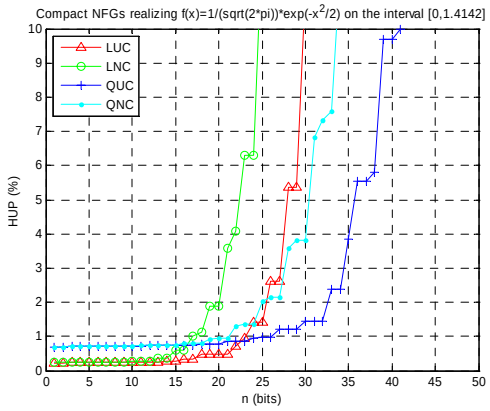
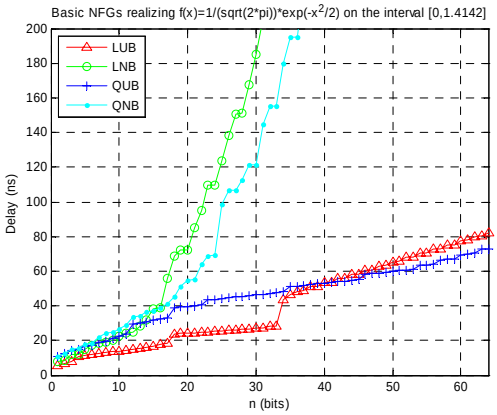
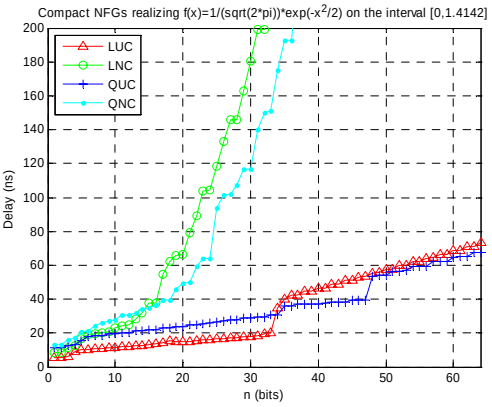
$f(x) = -x\log_2 x + (1-x)\log_2(1-x)$ on [1/256,1-1/256]

Basic Architectures	Compact Achitectures
	
	
	

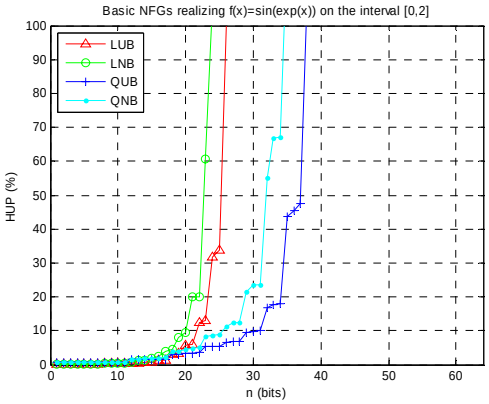
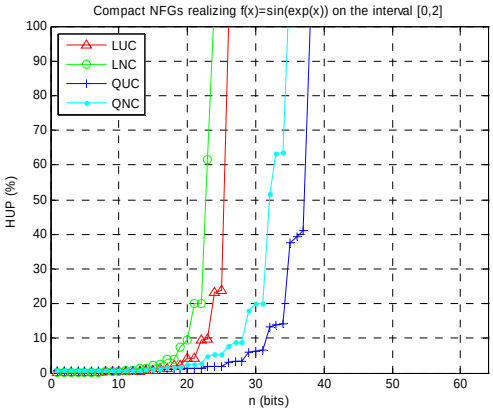
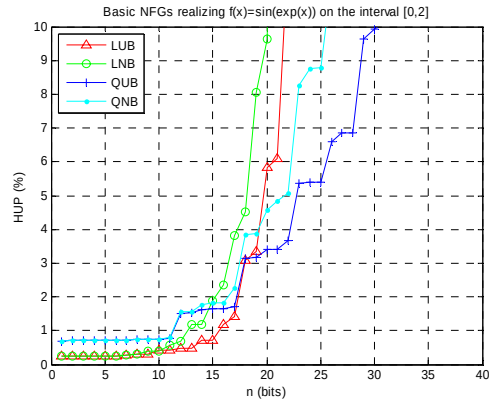
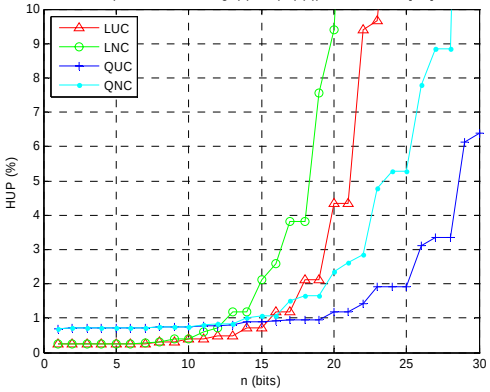
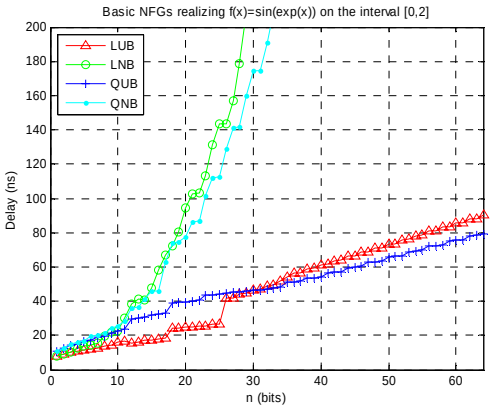
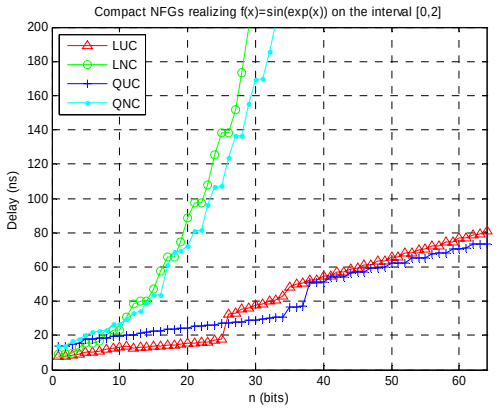
$$f(x) = \frac{1}{1+e^{-x}} \text{ on } [0,1]$$

Basic Architectures	Compact Achitectures
	
	
	

$$f(x)=\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}} \text{ on } \left[0,\sqrt{2}\right]$$

Basic Architectures	Compact Achitectures
	
	
	

$f(x) = \sin(e^x)$ on [0,2]

Basic Architectures	Compact Achitectures
	
	
	

D.2 THE BEST BASIC ARCHITECTURES FOR EACH FUNCTION

1. Based on Smallest HUP

Best Basic NFG based on HUP 1=LUB, 2=LNB,3=QUB, 4=QNB															
	Function Number														
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	2	2	1	1	1	1
2	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1
3	1	2	1	1	2	1	1	1	2	2	1	2	1	1	1
4	1	1	1	2	1	2	1	1	1	2	1	2	1	2	2
5	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
6	1	1	1	1	1	1	1	1	1	2	1	2	1	1	2
7	1	1	1	1	1	1	1	1	1	2	1	2	2	1	1
8	1	1	1	1	1	1	1	1	1	2	1	2	1	1	2
9	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
10	1	1	1	1	1	1	1	1	1	2	1	2	1	1	2
11	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
12	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
14	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	4	1	4	1	1	1
18	1	1	1	1	1	1	1	1	1	4	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	4	1	3	1	1	3
20	1	1	1	1	1	1	1	1	1	4	1	3	1	1	3
21	1	1	1	1	1	1	1	1	1	4	1	4	1	1	3
22	1	1	1	1	1	1	1	1	1	4	3	4	1	1	3
23	1	1	1	1	1	1	1	1	1	4	1	3	1	1	3
24	1	1	1	1	1	1	1	1	1	4	3	4	1	1	3
25	1	1	1	1	1	1	1	1	1	4	3	4	1	1	3
26	1	1	1	1	1	1	3	3	3	4	3	3	1	1	3
27	1	3	1	1	3	1	3	3	3	4	3	4	1	1	3
28	3	3	1	3	3	3	3	3	3	4	3	4	1	3	3
29	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
30	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
31	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
32	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
33	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
34	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
35	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
36	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
37	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
38	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
39	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
40	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
41	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
42	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
43	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
44	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
45	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
46	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
47	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
48	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
49	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
50	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
51	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
52	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
53	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
54	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
55	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
56	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
57	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
58	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
59	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
60	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
61	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
62	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
63	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
64	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3

2. Based on Shortest Delay

Best Basic NFG based on Delay 1=LUB, 2=LNB,3=QUB, 4=QNB															
	Function Number														
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	2	2	1	1	1	1
2	1	1	1	1	1	1	2	2	2	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
21	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
22	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
23	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
24	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
25	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
26	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
27	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
28	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
29	1	1	1	1	1	1	1	1	1	3	1	3	1	1	1
30	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
31	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
32	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
33	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
34	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
35	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
36	1	1	1	1	1	1	1	1	1	3	3	3	1	1	3
37	1	1	1	1	1	1	1	1	1	3	3	3	1	1	3
38	1	1	1	1	1	1	3	3	3	3	3	3	1	1	3
39	1	3	1	1	3	1	3	3	3	3	3	3	1	1	3
40	3	3	1	3	3	3	3	3	3	3	3	3	1	3	3
41	3	3	1	3	3	3	3	3	3	3	3	3	1	3	3
42	3	3	1	3	3	3	3	3	3	3	3	3	1	3	3
43	3	3	1	3	3	3	3	3	3	3	3	3	1	3	3
44	3	3	3	3	3	3	3	3	3	3	3	3	1	3	3
45	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
46	3	3	3	3	3	3	3	3	3	3	3	3	1	3	3
47	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
48	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
49	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
50	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
51	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
52	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
53	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
54	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
55	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
56	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
57	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
58	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
59	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
60	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
61	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
62	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
63	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
64	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

D.3 THE BEST COMPACT ARCHITECTURES FOR EACH FUNCTION VERSUS SIZE

1. Based on Smallest HUP

Best Compact NFG based on HUP 1=LUB, 2=LNB,3=QUB, 4=QNB															
	Function Number														
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
2	1	1	1	1	1	1	2	2	2	2	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
4	1	1	1	2	1	2	1	1	1	2	1	1	1	2	1
5	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
6	1	1	1	1	1	1	1	1	1	2	1	2	1	1	2
7	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
8	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
9	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
10	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1	2	1	2	1	1	1
12	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	4	1	4	1	1	1
16	1	1	1	1	1	1	1	1	1	4	1	4	1	1	3
17	1	1	1	1	1	1	1	1	1	4	1	4	1	1	3
18	1	1	1	1	1	1	1	1	1	4	1	4	1	1	3
19	1	1	1	1	1	1	1	1	1	4	1	3	1	1	3
20	1	1	1	1	1	1	1	1	1	4	3	3	1	1	3
21	1	1	1	1	1	1	1	1	1	4	3	4	1	1	3
22	1	1	1	1	1	1	3	3	3	4	3	4	1	1	3
23	3	3	1	3	3	3	3	3	3	4	3	3	1	3	3
24	3	3	3	3	3	3	3	3	3	4	3	4	1	3	3
25	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
26	3	3	3	3	3	3	3	3	3	4	3	3	3	3	3
27	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
28	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
29	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
30	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
31	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
32	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
33	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
34	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
35	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
36	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
37	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
38	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
39	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
40	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
41	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
42	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
43	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
44	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
45	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
46	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
47	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
48	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
49	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
50	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
51	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
52	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
53	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
54	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
55	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
56	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
57	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
58	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
59	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
60	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
61	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
62	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
63	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3
64	3	3	3	3	3	3	3	3	3	4	3	4	3	3	3

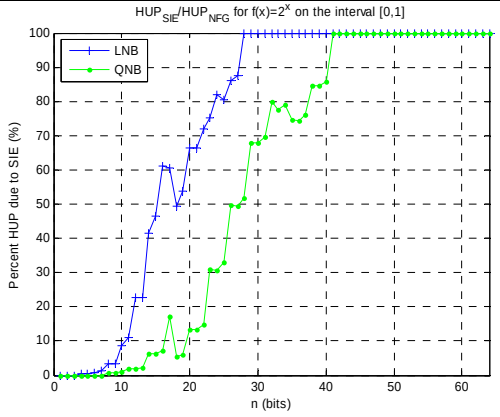
2. Based on Shortest Delay

Best Compact NFG based on Delay 1=LUB, 2=LNB,3=QUB, 4=QNB															
	Function Number														
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
21	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
22	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
23	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
24	1	1	1	1	1	1	1	1	1	3	1	1	1	1	1
25	1	1	1	1	1	1	1	1	1	3	1	3	1	1	1
26	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
27	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
28	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
29	1	1	1	1	1	1	1	1	1	3	1	3	1	1	3
30	1	1	1	1	1	1	1	1	1	1	3	3	1	1	3
31	1	1	1	1	1	1	1	1	1	1	3	3	1	1	3
32	1	1	1	1	1	1	3	3	3	3	3	3	1	1	3
33	1	3	1	1	3	1	3	3	3	1	3	1	1	1	3
34	3	3	1	3	3	3	3	3	3	3	3	1	1	3	3
35	3	3	1	3	3	3	3	3	3	3	3	3	1	3	3
36	3	3	3	3	3	3	3	3	3	3	3	1	1	3	3
37	3	3	3	3	3	3	3	3	3	1	3	1	3	3	3
38	3	3	3	3	3	3	3	3	3	3	3	1	3	3	3
39	3	3	3	3	3	3	3	3	3	1	3	1	3	3	3
40	3	3	3	3	3	3	3	3	3	3	3	1	3	3	3
41	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
42	3	3	3	3	3	3	3	3	3	3	3	1	3	3	3
43	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
44	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
45	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
46	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
47	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
48	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
49	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
50	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
51	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
52	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
53	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
54	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
55	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
56	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
57	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
58	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
59	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
60	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
61	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
62	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
63	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
64	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

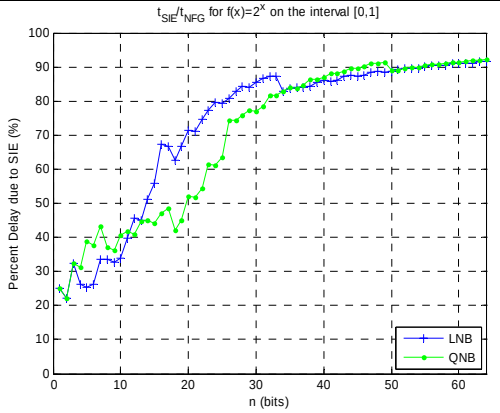
D.4 PERCENT HUP AND DELAY DUE TO SIE FOR LNB AND QNB NFGS

Basic architectures for $f(x) = 2^x$ on [0,1]

Hardware Utilization Percentage

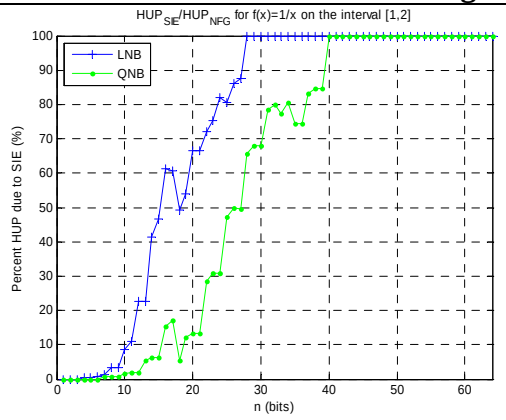


Combinational Delay

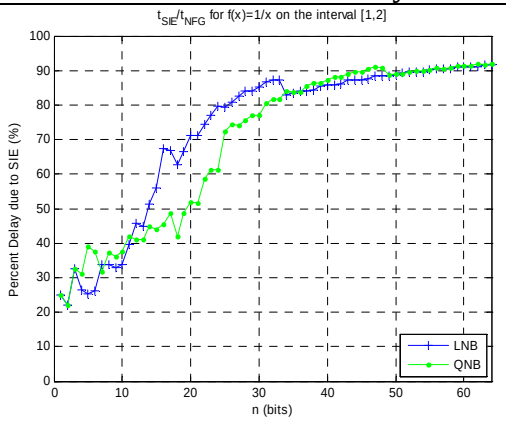


Basic architectures for $f(x) = 1/x$ on [1,2]

Hardware Utilization Percentage

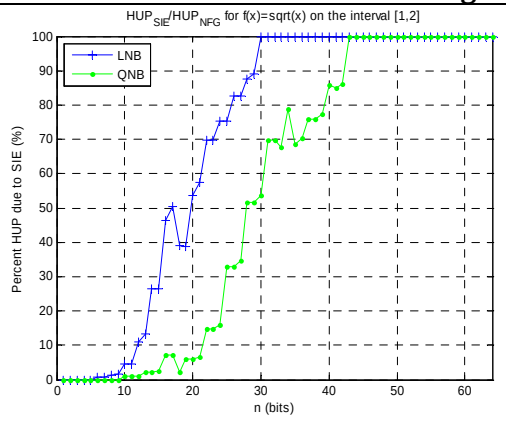


Combinational Delay

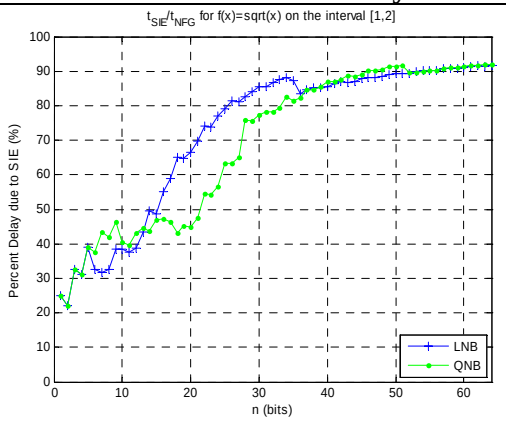


Basic architectures for $f(x) = \sqrt{x}$ on [1,2]

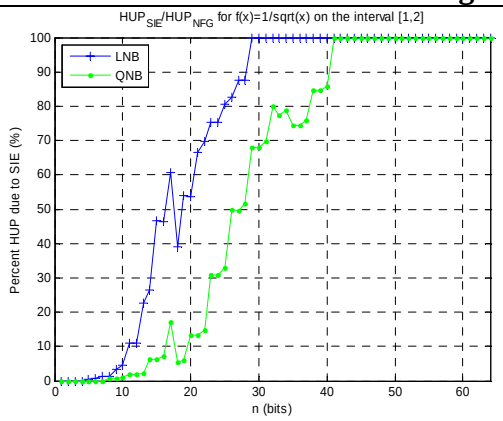
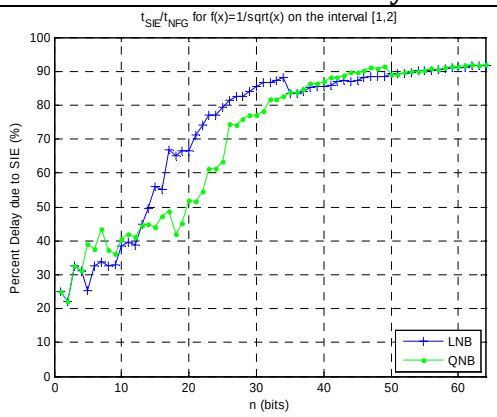
Hardware Utilization Percentage



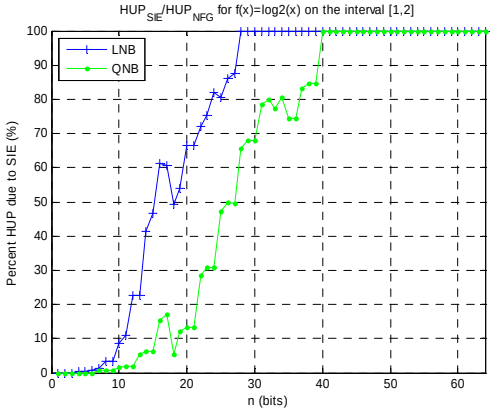
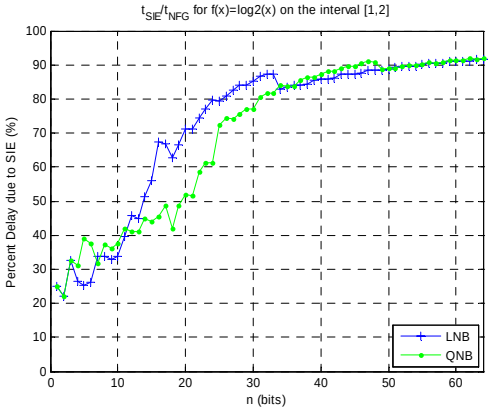
Combinational Delay



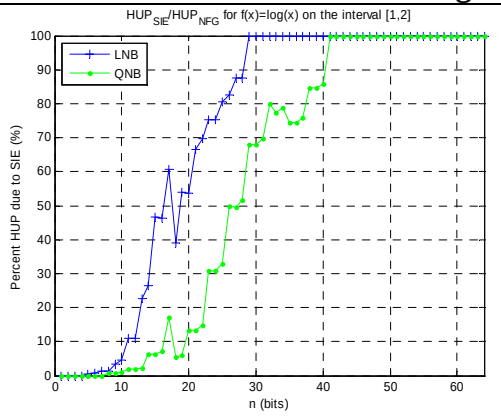
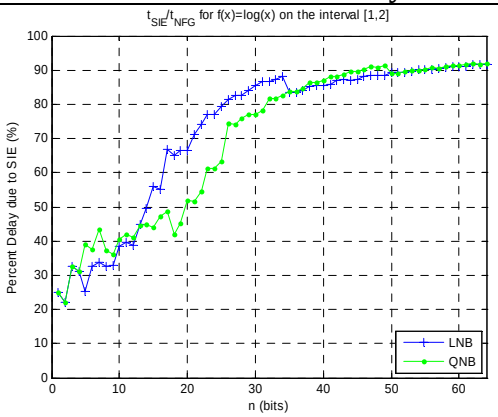
Basic architectures for $f(x) = 1/\sqrt{x}$ on [1,2]

Hardware Utilization Percentage	Combinational Delay
	

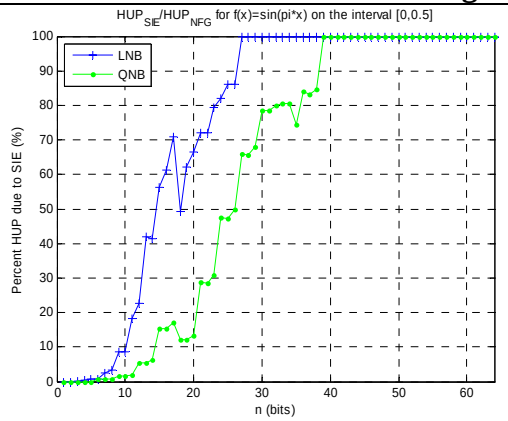
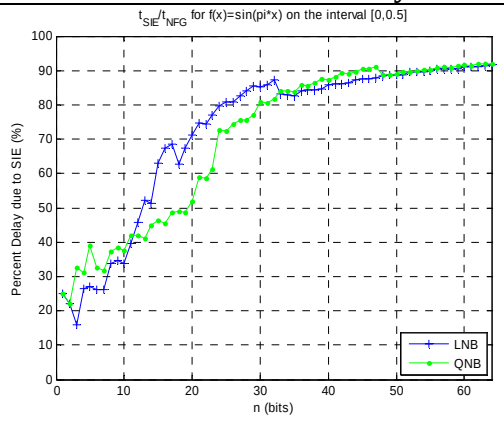
Basic architectures for $f(x) = \log_2 x$ on [1,2]

Hardware Utilization Percentage	Combinational Delay
	

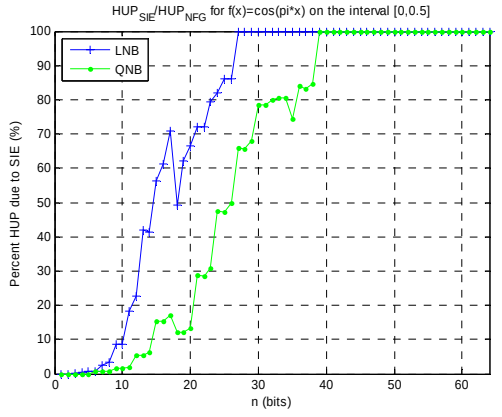
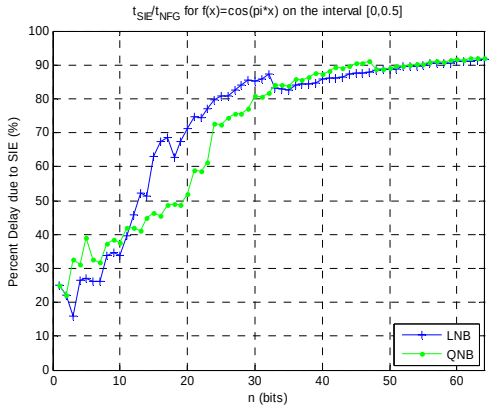
Basic architectures for $f(x) = \ln(x)$ on [1,2]

Hardware Utilization Percentage	Combinational Delay
	

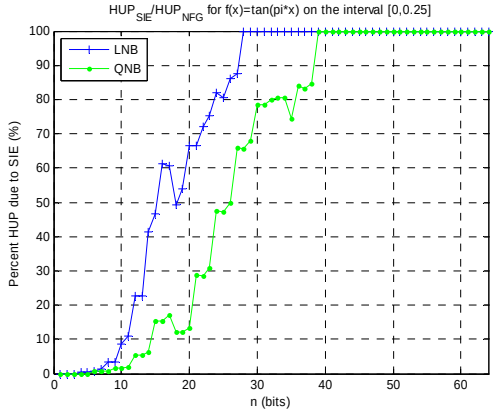
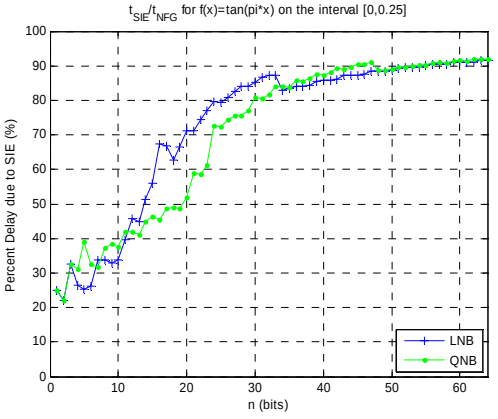
Basic architectures for $f(x) = \sin \pi x$ on [0,0.5]

Hardware Utilization Percentage	Combinational Delay
	

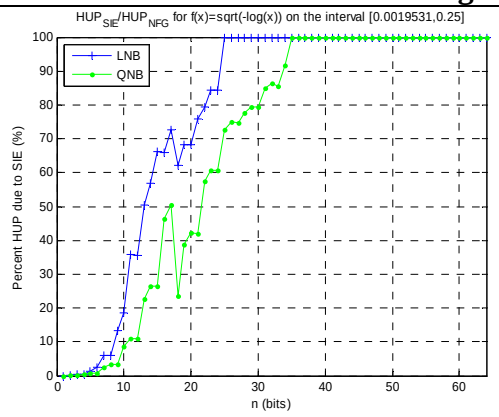
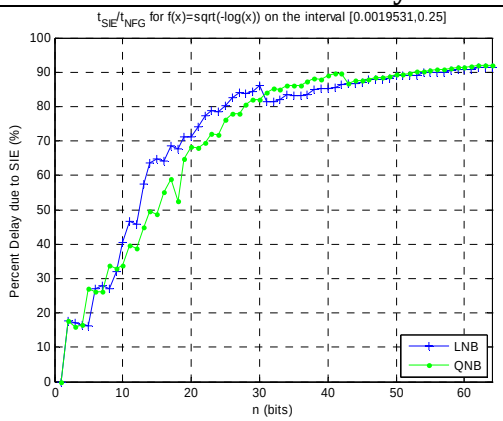
Basic architectures for $f(x) = \cos \pi x$ on [0,0.5]

Hardware Utilization Percentage	Combinational Delay
	

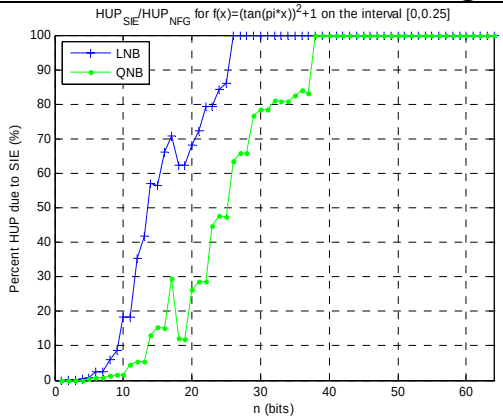
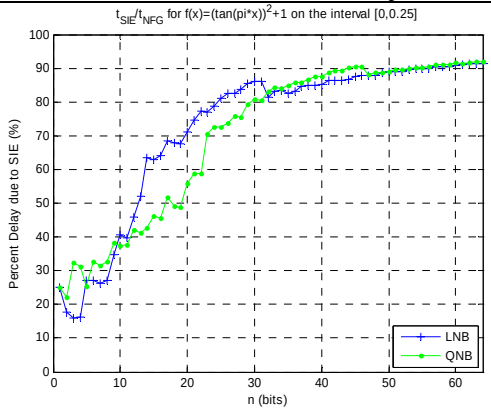
Basic architectures for $f(x) = \tan \pi x$ on [0,0.25]

Hardware Utilization Percentage	Combinational Delay
	

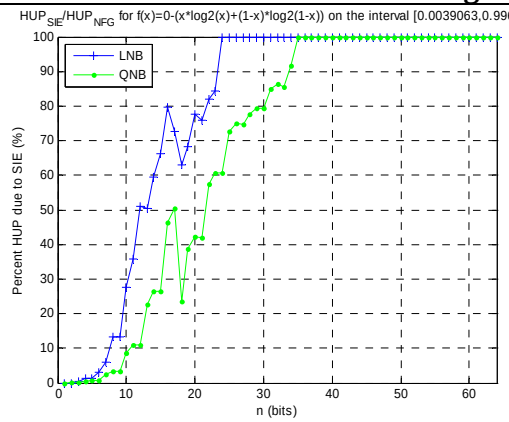
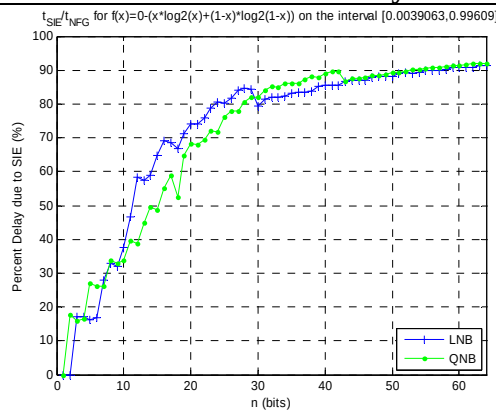
Basic architectures for $f(x)=\sqrt{-\ln x}$ on [1/512,1/4]

Hardware Utilization Percentage	Combinational Delay
	

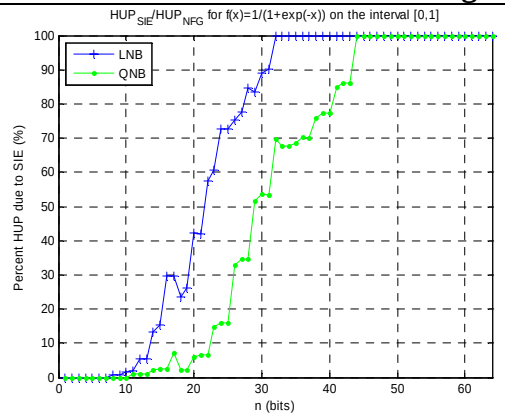
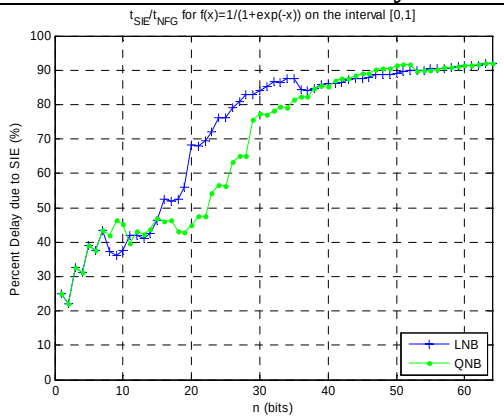
Basic architectures for $f(x)=\tan^2 \pi x+1$ on [0,0.25]

Hardware Utilization Percentage	Combinational Delay
	

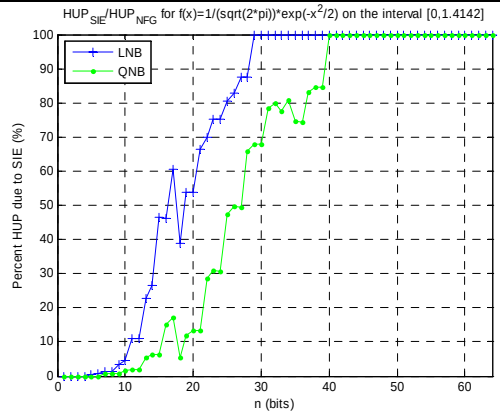
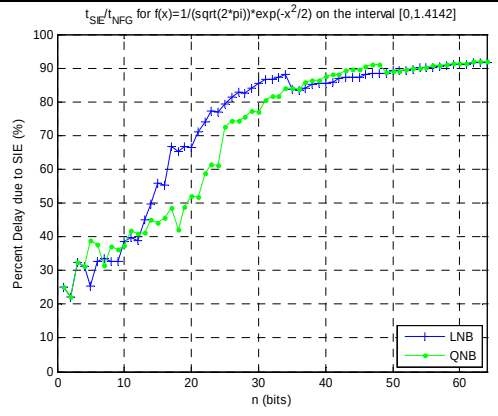
Basic architectures for $f(x)=-x\log_2 x+(1-x)\log_2(1-x)$ on [1/256,1-1/256]

Hardware Utilization Percentage	Combinational Delay
	

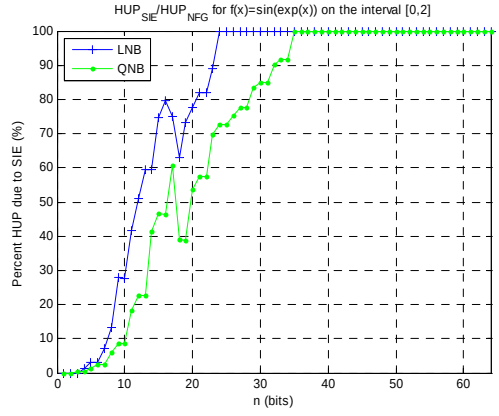
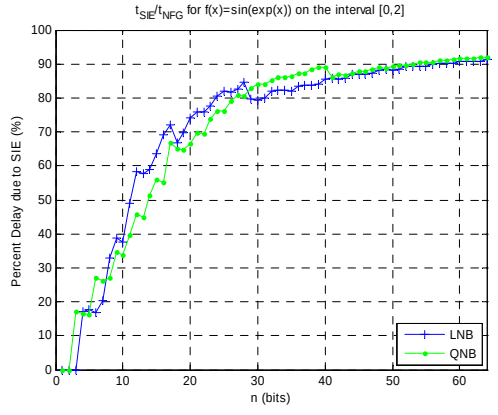
Basic architectures for $f(x)=\frac{1}{1+e^{-x}}$ on [0,1]

Hardware Utilization Percentage	Combinational Delay
	

Basic architectures for $f(x)=\frac{1}{\sqrt{2\pi}}e^{\frac{-x^2}{2}}$ on $\left[0,\sqrt{2}\right]$

Hardware Utilization Percentage	Combinational Delay
	

Basic architectures for $f(x)=\sin(e^x)$ on [0,2]

Hardware Utilization Percentage	Combinational Delay
	

LIST OF REFERENCES

[1] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers,' *Proc. Of the 1998 ACM/SIGDA Sixth Inter. Symp. on Field Programmable Gate Array (FPGA '98),* pp. 191-200, Monterey, CA, February 1998.

[2] J. Cao, B.W.Y. Wei, and J. Cheng, "High-performance architectures for elementary function generation," *IEEE Proceedings, 15th Symposium on Computer Arithmetic, 2001,* pp. 136-144.

[3] J. Cao, and B.W.Y. Wei, "High-performance hardware for function generation," *The 13th Symposium on Computer Arithmetic (ARITH '97),* pp. 184-189.

[4] C. L. Frenzen, T. Sasao, and J. T. Butler, "The tradeoff between memory size and approximation error in numeric function generators based on table lookup," preprint, February 25, 2007.

[5] C. L. Frenzen, N. Macaria, T. Sasao and J. T. Butler, "An efficient segmentation algorithm for the lookup table implementation of numeric function generators," preprint, July 2007.

[6] J.-M. Muller, *Elementary Function: Algorithms and Implementation,* Birkhauser Boston, Inc., Secaucus, NJ, 1997.

[7] S. Nagayama, T. Sasao, and J. T. Butler, "Compact numerical function generators based on quadratic approximation: Architecture and synthesis method", IEICE Trans. Fundamentals, (Special Section on VLSI Design and CAD Algorithms). Vol. E89-A, No. 12, pp. 3510-3518, December 2006.

[8] S. Nagayama, T. Sasao, and J. T. Butler, "Programmable numeric function generators based on quadratic approximation: architecture and synthesis method," *Asia and South Pacific Conference on Design Automation, 2006,* pp. 378-383, December 2006

[9] S. Nagayama, T. Sasao, and J. T. Butler, "Design method for numerical function generators using edge-valued binary decision diagrams," *IEICE Trans. Fundamentals,* Vol. E90-A, No. 12, December 2007.

[10] T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *The 12th Workshop on Synthesis And System Integration of Mixed Information Technologies 2004,* Kanazawa, Japan, October 18-19, 2004, pp. 422-429.

[11] T. Sasao, S. Nagayama, and J.T. Butler, "Numerical function generators using LUT cascades," *IEEE Trans. On Comp.,* Vol. 56, No. 6, pp 964-973, June 2007.

[12] T. Sasao, S. Nagayama, and J.T. Butler, “Programmable numerical function generators: architecture and synthesis method,” *Proc. Inter. Conf. on Field Programmable Logic and Applications (FPL ’05)*, Tampere, Finland, pp. 118-123, August 2005.

[13] M. Schulte, and E.E. Swartzlander, Jr., “Hardware designs for exactly rounded elementary functions,” *IEEE Trans. On Comp.*, Vol. 43, No. 8, pp 964-973, January 2007.

[14] Y. Iguchi, T. Sasao, and M. Matsuura, “Realization of multiple output functions by reconfigurable cascades,” *International Conference on Computer Designs: VLSI in Computers and Processors (ICCD’01)*, Austin, TX, pp.388-393, September 23-26, 2001.

[15] T. Sasao and M. Matsuura, “A method to decompose multiple-output logic functions,” *41st Design Automation Conference*, San Diego, CA, pp. 428-433, June 2-6, 2004.

[16] N. Takagi, T. Asada, and S. Yajima, “Redundant CORDIC Methods with a constant Scale Factor for Sine and Cosine Computation,” *IEEE Trans. Computers*, vol. 40, no. 9, pp.989-995, September 1991.

[17] J. E. Volder, “The CORDIC trigonometric computing technique,” *IRE Trans. Electronic Comput.*, Vol. EC-8, No. 3, pp. 330-334, September 1959.

[18] Xilinx Inc., “Virtex-II Platform FPGA: Complete Data Sheet,” DS031 (v3.4), Xilinx Inc., San Jose, CA, March 1, 2005.

[19] Xilinx Inc., “Virtex-II Platform FPGA: User’s Guide,” UG002 (v2.2), Xilinx Inc., San Jose, CA, November 5, 2007.

[20] N. Macaria, "High-speed numeric function generator using piecewise approximations,” M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 2007.

[21] J. F Wakerly, *Digital Design: Principles and Practices 4th Ed.*, Pearson Education, Inc., Upper Saddle River, NJ, 2006.

[22] T. Sasao and J.T. Butler, “Worst and best irredundant sum-of-products expressions,” *IEEE Trans. On Comp.*, Vol. 50, No. 9, September 2001.

[23] American National Standards Institute, “IEEE Standard 754 for binary floating point arithmetic,” *ANSI/IEEE Standard No. 754*, Washington DC, 1985.

[24] J. Stewart, *Calculus: Early Transcendentals 5th Ed.*, Thomson Learning, Inc., Belmont, CA, 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Prof. Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

5. Prof. Herschel H. Loomis
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Prof. Christopher L. Frenzen
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California

7. Prof. Douglas Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

8. LT Njuguna Macaria
   EDO School
   Port Hueneme, California

9. Mr. Alan Hunsberger
   National Security Agency
   Ft. Meade, Maryland

10. Dr. John Harkins
National Security Agency
Ft. Meade, Maryland

11. Dr. Pedro Claudio
Staff Systems Engineer
Technical Operations Applied Research
Lockheed Martin Missiles and Fire Control
Orlando, Florida

12. Mr. David Caliga
SRC Computers
Colorado Springs, Colorado

13. Mr. Jon Huppenthal
SRC Computers
Colorado Springs Colorado

14. LT Tim Knudstrup
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California